AD-A031 429 ILLINOIS UNIV AT URBANA-CHAMPAIGN COORDINATED SCIENCE LAB F/G 9/2
THE DESIGN OF ALTERNATING LOGIC SYSTEMS WITH FAULT DETECTION CA--ETC(U)
AUG 76 D A REYNOLDS DAAB07-72-C-0259
UNCLASSIFIED R-738 NL

1 OF 2
AD
A031429

FG

REPORT R-738 AUGUST, 1976 UILU-ENG 76-2226

CSL COORDINATED SCIENCE LABORATORY

12

THE DESIGN OF ALTERNATING LOGIC SYSTEMS WITH FAULT DETECTION CAPABILITIES

DENNIS ANDREW REYNOLDS

UNIVERSITY OF ILLINOIS - URBANA, ILLINOIS

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
~~4. TITLE (and Subtitle)~~ 6 THE DESIGN OF ALTERNATING LOGIC SYSTEMS WITH FAULT DETECTION CAPABILITIES.		5. TYPE OF REPORT & PERIOD COVERED Technical Report
		~~6. PERFORMING ORG. REPORT NUMBER~~ 14 R-738, UILU-ENG-76-2226 ✓
7. AUTHOR(s) 10 Dennis Andrew Reynolds		~~8. CONTRACT OR GRANT NUMBER(s)~~ 15 DAAB-07-72-C-0259
9. PERFORMING ORGANIZATION NAME AND ADDRESS Coordinated Science Laboratory ✓ University of Illinois at Urbana-Champaign Urbana, Illinois 61801		10. PROGRAM ELEMENT, PROJECT, TASK ~~AREA & WORK UNIT NUMBERS~~ 12 149 P.
11. CONTROLLING OFFICE NAME AND ADDRESS Joint Services Electronics Program		~~12. REPORT DATE~~ 11 Aug~~ust,~~ 1976
		~~13.~~ NUMBER OF PAGES 140
~~14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)~~ 9 Doctoral thesis,		15. SECURITY CLASS. (of this report) UNCLASSIFIED
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE
16. DISTRIBUTION STATEMENT (of this Report) Approved for public release; distribution unlimited		
17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)		
18. SUPPLEMENTARY NOTES		

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Alternating Logic	Stuck-At-Faults	Time Redundancy
Alternating Systems	Totally Self-Checking	Alternating Logic Primitives
Single Fault	Combinational Network	
On-Line Detection	Synchronous Machine	

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

The fault detection capability of a design technique named "alternating logic design" is detailed. The technique achieves its fault detection capability utilizing a redundancy in time instead of the more conventional space redundancy and is based on the successive execution of a required function and its dual. In combinational networks the method involves the utilization of a self-dual function to represent the required function and the realization of the self-dual function in a network with structural properties which are sufficient to guarantee the detection of all single

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

097 700

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

dn

20. ABSTRACT (continued)

faults are derived.

The application of alternating logic design to synchronous sequential machines is also studied, and a method for realizing any synchronous sequential machine with feedback and output circuitry designed using combinational alternating logic techniques is presented. The resulting realization is capable of detecting all single faults in the feedback and output combinational circuitry and all single input and output faults associated with the memory elements.

Alternating logic design using alternating modules as basic design elements is also examined. Necessary conditions that a set of alternating modules be sufficient to construct alternating networks representing any arbitrary Boolean function are derived.

UILU-ENG 76-2226

THE DESIGN OF ALTERNATING LOGIC SYSTEMS WITH FAULT DETECTION CAPABILITIES

by

Dennis Andrew Reynolds

This work was supported in part by the Joint Services Electronics Program (U.S. Army, U.S. Navy and U.S. Air Force) under Contract DAAB-07-72-C-0259.

THE DESIGN OF ALTERNATING LOGIC SYSTEMS WITH FAULT DETECTION CAPABILITIES

BY

DENNIS ANDREW REYNOLDS

B.S.E.E., University of Arkansas, 1969
M.S., University of Illinois, 1970

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Electrical Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1976

Thesis Advisor: Professor Gernot Metze

Urbana, Illinois

THE DESIGN OF ALTERNATING LOGIC SYSTEMS WITH FAULT DETECTION CAPABILITIES

Dennis Andrew Reynolds, Ph.D.
Coordinated Science Laboratory and
Department of Electrical Engineering
University of Illinois at Urbana-Champaign, 1976

The fault detection capability of a design technique named "alternating logic design" is detailed. The technique achieves its fault detection capability utilizing a redundancy in time instead of the more conventional space redundancy and is based on the successive execution of a required function and its dual. In combinational networks the method involves the utilization of a self-dual function to represent the required function and the realization of the self-dual function in a network with structural properties which are sufficient to guarantee the detection of all single faults. Necessary and sufficient structural properties for any alternating network to be capable of detecting all single faults are derived.

The application of alternating logic design of synchronous sequential machines is also studied, and a method for realizing any synchronous sequential machine with feedback and output circuitry designed using combinational alternating logic techniques is presented. The resulting realization is capable of detecting all single faults in the feedback and output combinational circuitry and all single input and output faults associated with the memory elements.

Alternating logic design using alternating modules as basic design elements is also examined. Necessary conditions that a set of alternating modules be sufficient to construct alternating networks representing any arbitrary Boolean function are derived.

ACKNOWLEDGMENT

The author would like to express his gratitude to his thesis advisor, Professor Gernot Metze, for his guidance afforded the author in the production of this thesis. Further, the author would also like to thank his colleagues of the Switching Systems Group at the Coordinated Science Laboratory for helping to create a congenial atmosphere conducive to productive research and to thank the Coordinated Science Laboratory staff, particularly Mrs. Rose Harris, for their assistance in producing the manuscript.

The author also thanks Sandia Laboratories for supporting the author financially thus making the production of the thesis possible.

Finally, the author would like to dedicate the thesis to his father, Ira E. Reynolds, who enabled the author to complete an undergraduate engineering curriculum, giving up personal luxuries for himself in extending the necessary financial support. The support and patience extended the author by his wife, Beverly, is also gratefully appreciated.

TABLE OF CONTENTS

TABLE OF CONTENTS (continued)

1. INTRODUCTION

1.1. Current Approaches to Fault Tolerant Digital Systems

In recent years due to the prevalent use of integrated circuits the complexity and size of digital systems has increased dramatically. As a result system reliability has become a topic of major concern. And, although integrated circuit manufacturers make efforts to ensure the reliability of their products, the size of digital systems today makes the probability of a circuit fault somewhere in a complex digital system at any given time a factor which must be considered. Considerable research has, therefore, been conducted in order to find techniques for detecting system faults. One method for doing this in programmable digital systems is periodic software checking.

Periodic software checking essentially involves the utilization of diagnostic programs at periodic intervals as a means to test a programmable digital system for correct operation. If a system fault is detected during this testing period then necessary repair or replacement is made. Obviously, such a method can be used only where human intervention is possible. There, however, are several drawbacks associated with periodic software checking. First, the diagnostic programs written for complex digital systems seldom fully check the system for correct operation. This is particularly true if the system is already built. If the diagnostic routines are developed concurrently with the system design, then a more diagnosable and maintainable system generally results, and initial system checkout time is usually reduced. But still, the system is seldom fully checked. Second, during the testing or checking period when the diagnostic routines are running, valuable system time is lost. This lost time can be very significant for more complex digital systems. Third, since accurate results are not assured until the testing period following the computation, such a digital system cannot be used in a critical application where accuracy of system output must be guaranteed. Also, even if testing reveals no faults, the correctness of prior system output cannot be fully assured since transient faults occurring outside of the testing period can cause erroneous results which will not be detected. Fourth, if a fault is detected during the testing period, then all system output obtained since the last testing period must be suspect. Thus, when a checkout fails, computation must be rolled back to the previous testing period. The effect is lost time and money.

A second approach to fault tolerant digital systems is the fault masking redundancy approach. With this approach, a digital system is designed to operate with high reliability for a period of time and is then expected to fail, the failure occurring when enough faults accumulate to exceed the protection designed into the system through redundancy. Such a system is suitable for applications on space missions where reliable operation is required only for the mission duration. Application of the approach to maintained digital systems is not too suitable as the fault masking redundancy built into the system also makes the system difficult to test. Triple Modular Redundancy (TMR) [11,16] and Quadded Logic [11,16] are probably the most common fault masking redundancy techniques used.

A third approach to the design of fault tolerant digital systems, self-checking design, overcomes some of the disadvantages of periodic software checking and fault masking redundancy techniques. The main advantage of the design scheme is the immediate detection of faults which affect system operation. The appearance of these faults is indicated using hardware monitors which continuously check for improperly encoded system information. One area of application for such self-checking digital systems has been in telephone switching computers, e.g. the No. 1 ESS [15] and the No. 2 ESS [6]. In the No. 1 ESS information was encoded in a single error correcting, double error detecting Hamming code. The No. 2 ESS used a simple parity code for information encoding. In these computers any improperly encoded information produced by a module was detected by the hardware monitors, and a duplicate of the failed module was switched in automatically, thus allowing repair of the faulty module off-line with uninterrupted computer operation. Another self-checking computer, the Self-Testing and Repairing (STAR) computer [3] was developed and tested at the Jet Propulsion Laboratory. The STAR computer utilized self-checking circuits in a triple modular system containing spares. Defective module operation detected by the Test and Repair Processor (TARP) of the computer initiated the replacement of the faulty module by one of the standby spares. The TARP itself was triplicated and was considered the hardcore of the system.

Another type of self-checking digital system, the <u>totally self-checking</u> digital system was introduced by Carter and Schneider [9]. Such digital systems employ on-line fault detection and guarantee that no erroneous data are passed to the user without an error indication for all faults from a prescribed set. The design procedures detailed in this thesis produce totally self-checking digital systems. A literature survey on previous work in totally self-checking digital systems is presented in the following section.

1.2. Literature Survey on Totally Self-Checking Digital Systems

Totally self-checking (TSC) digital system design has been studied in some detail since the design problem was formalized in Carter and Schneider's [9] introductory paper. The approach has been to consider a digital system as composed of interconnections of functional modules, Anderson [1], each of which is monitored by a hardware checker. Inputs and outputs to each functional unit are encoded, and faulty functional module operation is indicated by the production of an improperly encoded module output. This improper output is detected by a TSC checker, and the erroneous output is indicated on the checker output. Design of the TSC checker, however, has presented some problems.

Carter and Schneider first addressed the problem of TSC checker design. Included in their introductory paper [9] were network diagrams for 3 TSC checkers; a parity code checker, a 2-out-of-5 code checker, and a two-rail code checker. The checkers were, however, assumed to be self-checking only for faults on gate outputs, not on both checker gate outputs and inputs. Anderson [1] modeled faults on both gate outputs and inputs and presented several totally self-checking two-rail checker designs as well as parity checker designs. Anderson and Metze [2] also investigated m-out-of-n checkers in general and formulated a checker design for the k-out-of-2k codes. Reddy [28] extended Anderson and Metze's work on the k-out-of-2k checker by incorporating an easily testable cellular array [30] in the design. The result was a k-out-of-2k checker with a smaller test set than that of Anderson and Metze. Design rules for the general m-out-of-n checker were formulated later by Smith and Metze [33]. Further, the design of two level m-out-of-n checkers was shown to be related to the Ramsey number problem.

Research on TSC checkers for other classes of codes has been minimal. A TSC checker for the Berger code [5] has been reported by Ashjahee and Reddy [29]. The checker, however, utilizes a code-disjoint translator to translate the Berger code into a 1-out-of-n code. That code is then fed into a 1-out-of-n self-checking checker.

Design of the self-checking functional unit whose operation is monitored by the TSC checker has received less attention than the TSC checker design itself. A significant amount of work, however, has been done on synchronous machine design, particularly on fail-safe design. Some of the machine designs resulting from this work have properties suitable for application in TSC designs. Thoma [34] used a symmetric fault model and employed on-set and off-set form equations with a k-out-of-n code for state variable encoding. Thoma's method was later modified by Diaz [13] so that only on-set form equations were used in the design. Formally pointing out the relationships between some fail-safe designs and totally self-checking designs, Diaz [14] extended TSC design concepts to Moore-synchronous machines using a two-rail code for the input and output variables and a k-out-of-n code for the state variables.

Recently, Sawin [31] proposed another method for fail-safe synchronous machine design using the on-set realization only. Sawin's approach was drastically different from others because his state assignment was tailored specifically for the given flow table. A further improvement of the on-set realization method was presented in Wang's [35] paper. Wang's approach was similar to that of Sawin's but led to a generalization of the method.

Much less work has been done on TSC asynchronous sequential machines. Recent work by Ozguner [26], however, has led to TSC asynchronous machine design methods. Other authors [23,27] have proposed designs, but none have been totally self-checking.

All the methods for TSC checker and functional module design discussed thus far have one property in common. All use space encodings to achieve the required measure of fault detection capability. There exists, however, an alternate encoding scheme, time encoding, which achieves similar results. A type of time encoding, the two-rail time encoding, is the subject of this thesis. Systems utilizing this type of time encoding are called alternating systems or alternating logic systems. Such systems were first proposed by Bark and Kinne [4] in 1953. No reference, however, was made to the fault detecting capabilities of the method until Yamamoto [36] et al. published an introductory paper on the subject. This thesis utilizes the fault detection capabilities of alternating logic in the design of totally self-checking alternating logic systems.

2. SELF-CHECKING COMBINATIONAL ALTERNATING LOGIC

2.1. Combinational Alternating Logic Design

Combinational alternating logic design is a design technique resulting in combinational alternating networks. Any input to an alternating network used within an alternating system or which is in itself an alternating system must be an alternating binary variable. Alternating binary variable, alternating network, and alternating logic design are thus related terms. Alternating binary sequence is first defined.

Definition 2.1: An alternating binary sequence is a sequence of symbols from the set $\{0,1\}$ admitting to a sequential parsing into 2-tuples of the form $(d,\bar{d})$.

For example, the sequence 011001011001 is an alternating binary sequence as the parsing 01/10/01/01/10/01 can be made. The sequence 011101100110, however, is not.

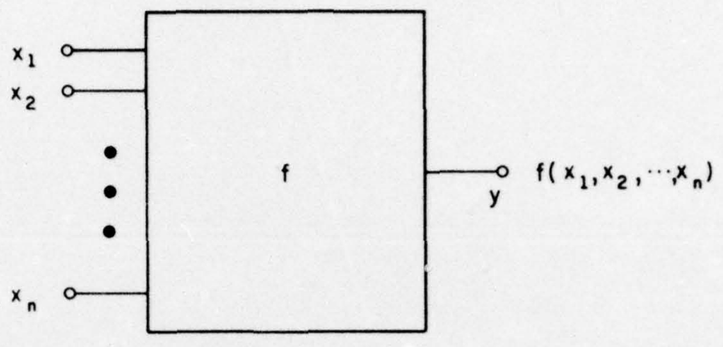
Definition 2.2: An alternating binary variable is a variable which assumes values in 2-tuples from the set $\{01,10\}$.

As a result, any sequence produced by an alternating binary variable when viewed as a binary sequence is an alternating binary sequence.

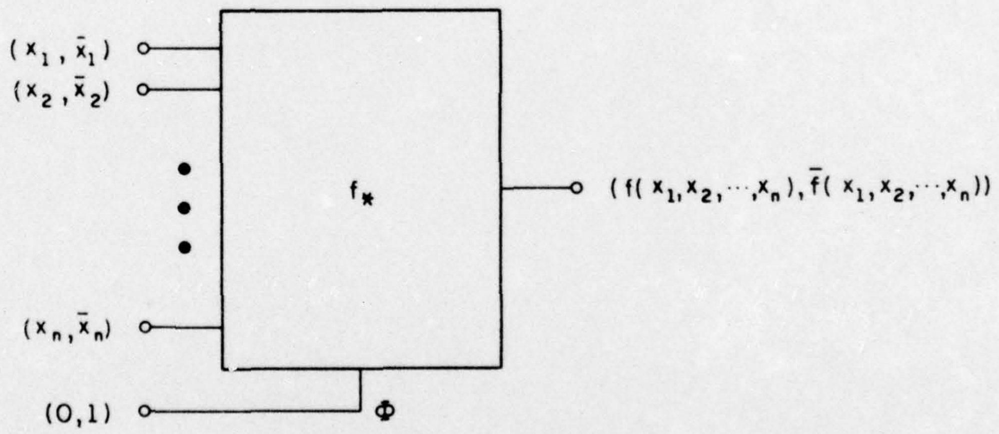
Now, consider a combinational network, Figure 2.1 a), with n inputs $x_1,x_2,\ldots,x_n$ and one output y realizing a function f. That is,

$$y = f(x_1,x_2,\ldots,x_n) = f(X),$$

where X is a binary n-vector. If we assume that $x_1,x_2,\ldots,x_n$ are alternating



a) Combinational Network Realizing f



b) Alternating Combinational Network Representing f

FP- 5045

Figure 2.1. Combinational alternating logic network model.

binary variables and that they alternate in synchronism, then input vectors, X, are applied to the network realizing f in complementary pairs. The network input can therefore be represented as a vector 2-tuple, $(X,\overline{X})$ where

$$(X,\overline{X}) = (x_1x_2\ldots x_n,\overline{x}_1\overline{x}_2\ldots\overline{x}_n).$$

The output produced as a result of this input must also be a 2-tuple although not necessarily complementary. Since the network realizes f the output 2-tuple must be

$$(f(x_1,x_2,\ldots,x_n),\ f(\overline{x}_1,\overline{x}_2,\ldots,\overline{x}_n))$$

or

$$(f(X),f(\overline{X})).$$

Definition 2.3: A network realizing the function f is an alternating network iff for every $(X,\overline{X})$ an output $(f(X),f(\overline{X}))$ is produced such that $f(\overline{X}) = \overline{f}(X)$.

In other words, the network realizing f is an alternating network if the output variable y alternates when synchronized alternating variables are applied as inputs. Further, since y is produced as a result of the application of $(X,\overline{X})$, y alternates in synchronism with $(X,\overline{X})$. Definition 2.3 thus places a significant restriction on the functions realized by alternating networks. This class of functions is defined in the following theorem.

Theorem 2.1: A network realizing a function f is an alternating network iff f is a self-dual function.

Proof: If f is self-dual then $f(\overline{X}) = \overline{f}(X)$ for every X and any network realizing f must therefore be an alternating network. Conversely, if a network is an alternating network then $f(\overline{X}) = \overline{f}(X)$ and f is self-dual.

Q.E.D.

Essentially, Theorem 2.1 says that only self-dual functions can be realized by networks that are alternating networks. However, consider a function f that is not self-dual and define a function f_* of n+1 variables $\Phi, x_1, x_2, \ldots, x_n$ as follows. Let

$$f_*(0,x_1,x_2,\ldots,x_n) = f_*(0,X) \overset{\Delta}{=} f(X)$$

and

$$f_*(1,x_1,x_2,\ldots,x_n) = f_*(1,X) \overset{\Delta}{=} \bar{f}(\bar{X})$$

for every X. In a sense $f_*(\Phi,x_1,x_2,\ldots,x_n)$ represents $f(x_1,x_2,\ldots,x_n)$ since $f_*(0,x_1,x_2,\ldots,x_n) = f(x_1,x_2,\ldots,x_n)$. Further, f_* is a self-dual function as shown in the following theorem.

<u>Theorem 2.2</u>: The function f_* of n+1 variables $\Phi, x_1, x_2, \ldots, x_n$ defined by

$$f_*(0,x_1,x_2,\ldots,x_n) = f(x_1,x_2,\ldots,x_n)$$

$$f_*(1,x_1,x_2,\ldots,x_n) = \bar{f}(\bar{x}_1,\bar{x}_2,\ldots,\bar{x}_n)$$

is self-dual for arbitrary f.

<u>Proof</u>: Using Shannon's expansion theorem $f_*(\Phi,X)$ can be written as

$$\begin{aligned} f_*(\Phi,X) &= \Phi f_*(1,X) + \bar{\Phi} f_*(0,X) \\ &= \Phi \bar{f}(\bar{X}) + \bar{\Phi}(\quad(X). \end{aligned}$$

But then

$$\begin{aligned} \bar{f}_*(\bar{\Phi},\bar{X}) &= \overline{\bar{\Phi}\;\bar{f}(X) + \Phi\, f(\bar{X})} = (\overline{\bar{\Phi}\;\bar{f}(X)})(\overline{\Phi\, f(\bar{X})}) \\ &= (\Phi + f(X))(\bar{\Phi} + \bar{f}(\bar{X})) \\ &= \Phi \bar{f}(\bar{X}) + f(X)\bar{\Phi} + f(X)\bar{f}(\bar{X}). \end{aligned}$$

Thus, by consensus

$$\overline{f}_*(\overline{\Phi},\overline{X}) = \Phi\overline{f}(\overline{X}) + \overline{\Phi}f(X) = f_*(\Phi,X),$$

and $f_*(\Phi,x_1,x_2,\ldots,x_n)$ is self-dual. Q.E.D.

In an alternating system it is the function f_* which is used to realize a function f whenever f is not self-dual. In fact f_* is called the dualization of f. The Φ input variable of f_* becomes a clock and is synchronized to the input variables $(X,\overline{X})$. That is, $\Phi = 0$ when X is applied and $\Phi = 1$ when $\overline{X}$ is applied. Thus, when $\Phi = 0$

$$f_*(0,x_1,x_2,\ldots,x_n) = f(x_1,x_2,\ldots,x_n),$$

and when $\Phi = 1$

$$f_* = f_*(1,\overline{x}_1,\overline{x}_2,\ldots,\overline{x}_n) = \overline{f}_*(0,x_1,x_2,\ldots,x_n) = \overline{f}(x_1,x_2,\ldots,x_n).$$

With the (0,1) clock Φ applied as input the self-dual function f_* thus produces an alternating output $(f(X),\overline{f}(X))$ representing f. The network f_* is modeled in Figure 2.1 b). Alternating logic design can now be defined.

Definition 2.4: Alternating logic design: is a design technique whereby combinational networks are represented utilizing self-dual functions or compositions of self-dual functions, and system input variables are constrained to be alternating variables.

Alternating logic design is therefore implemented by first formulating a functional description of the required function, f. If f is self-dual any network realizing f is an alternating network and alternating input variables $(x_1,\overline{x}_1),(x_2,\overline{x}_2),\ldots,(x_n,\overline{x}_n)$ are applied directly to that network so as to produce $(f(X),\overline{f}(X))$. If f is not self-dual, a functional description of f_* is generated, and f_* is realized in a conventional manner. Then, a

(0,1) clock is applied to the Φ input, and alternating variables $(x_1,\overline{x}_1)$, $(x_2,\overline{x}_2),\ldots,(x_n,\overline{x}_n)$ synchronized to the (0,1) clock are applied as inputs resulting in an alternating output $(f(X),\overline{f}(X))$ from the network realizing f_*. The design technique is illustrated by the following example.

Example 2.1: Consider the three variable functional description given in Figure 2.2 a) for the function f. The problem is to design an alternating network which represents f. Since $f(1,0,1) \neq \overline{f}(0,1,0)$, f is not self-dual. So, any network realizing f is not an alternating network. However, by adding a fourth variable, Φ, and dualizing f, a functional description of f_*, Figure 2.2 b), is obtained. From Figure 2.2 b),

$$f_*(\Phi,x_1,x_2,x_3) = \overline{\Phi}\,\overline{x}_2 + \overline{\Phi}\,\overline{x}_1\overline{x}_3 + \overline{x}_2\overline{x}_3 + \overline{x}_1\overline{x}_3$$

$$\Rightarrow f_* = \overline{(\Phi + x_2)} + \overline{(\Phi + x_1 + x_3)} + \overline{(x_2 + x_3)} + \overline{(x_1 + x_3)}$$

$$\Rightarrow f_* = \overline{(\Phi + x_2)(\Phi + x_1 + x_3)(x_2 + x_3)(x_1 + x_3)}.$$

The alternating network representing f is shown in Figure 2.2 c).

2.2. Single Fault Model

All digital networks, alternating networks included, are susceptible to physical failures which affect network operation in one way or another. Several models to represent these network failures have been used, the most popular of which is the line fault model. One line fault model, Carter and Schneider [9], assumes failures can be represented as gate output lines stuck-at-1 (s-@-1) or stuck-at-0 (s-@-0). Another more general line fault

x_1 \ $x_2 x_3$	00	01	11	10
0	1	1	0	1
1	1	1	0	0

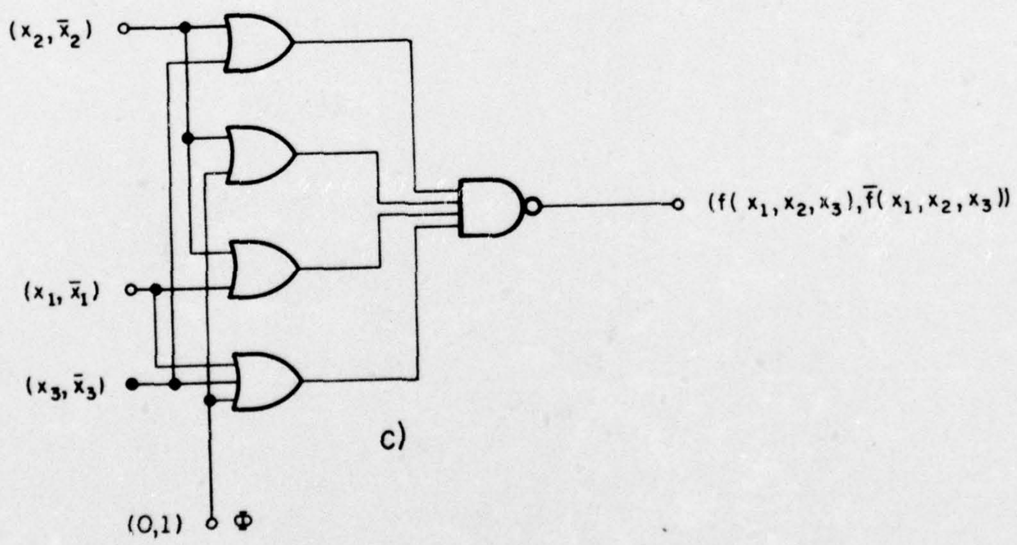
$f(x_1, x_2, x_3)$

a)

Φx_1 \ $x_2 x_3$	00	01	11	10
00	1	1	0	1
01	1	1	0	0
11	1	0	0	0
10	1	1	0	0

$f_*(\Phi, x_1, x_2, x_3)$

b)



c)

Figure 2.2. Dualization procedure, Example 2.1.

model, Anderson and Metze [2], assumes all failures to be representable by line faults on all gate input and output lines, s-@-0 and s-@-1. The latter representation will be used here. Thus, the term _fault_ will refer to a network condition in which one or more gate input or output lines are s-@-0 or s-@-1. The term _single fault_ will refer to a network condition in which only one gate input or one gate output line is s-@-0 or s-@-1. A fault involving several lines will be termed a _multiple fault_. A _unidirectional fault_ will be defined as a multiple fault in which the faulty lines are all stuck at the same logical value, either all s-@-0 or all s-@-1.

Of particular interest in relation to alternating networks is the set of all single line faults. In fact, alternating networks will be shown to be capable of detecting all single faults on gate input and output lines, s-@-0 or s-@-1, during normal network operation while assuring error-free operation prior to that detection. Continuous error-free operation, however hinges on several assumptions. First, it is assumed that initially the network is free of all faults. The means by which this is guaranteed is not specified. It is also assumed that the most probable network fault is the single fault and that the occurrence of the first fault is detected prior to the occurrence of a second single fault during normal network operation. This allows for repair of the network so that detection of the second single fault can be guaranteed. Under these assumptions at most one single fault is present in the network at any given time and continuous error-free operation is assured.

2.3. Self-Checking Property

Consider an irredundant combinational alternating network N with n inputs $(x_1,\overline{x}_1),(x_2,\overline{x}_2),\ldots,(x_n,\overline{x}_n)$ and one output y which represents a function f. Assume that a network fault of some type occurs. For a particular input $(X,\overline{X})$ either the fault does not affect network operation, and the correct alternating output is produced, say $(d,\overline{d})$, or one of three possible errors occurs. If the fault affects N under X only, $(\overline{d},\overline{d})$ is produced; if the fault affects N under $\overline{X}$ only, then (d,d) is produced; and if the fault affects N under both X and $\overline{X}$, then $(\overline{d},d)$ is produced. Since in the fault-free state the network output alternates for all inputs, errors of the first two types are considered to be detectable, whereas an error of the last type is considered to be undetectable. The network output for this case is termed an erroneous alternating output.

<u>Definition 2.5</u>: A network is <u>self-checking</u> for all faults from a prescribed set, F, if for every fault from F there exists at least one input $(X,\overline{X})$ which produces a detectable error on the network output, and there exists no input $(X,\overline{X})$ which produces an erroneous alternating output.

If F is constrained to be the set of all single faults of the stuck-at-0 or stuck-at-1 type on all primary network inputs and all network lines, then it is possible to derive necessary and sufficient conditions that the alternating network realizing f be self-checking. But, to do this some definitions must first be made.

<u>Definition 2.6</u>: A <u>network line</u> is any line in a network other than a primary network input. (Branch lines from fanout points are considered to be distinct network lines.)

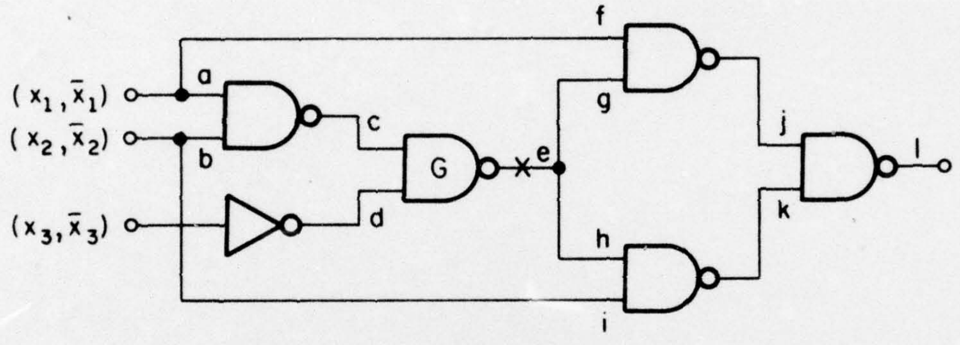
Network lines are labeled with small letters and a fault on a line is denoted by f_a^d where the subscript is the line label, and the superscript is the logical value at which the line is stuck. For example, in Figure 2.3 an alternating network representing the function

$$f = x_1x_2 + x_1x_3 + x_2x_3$$

is shown. Network lines in the circuit are labeled a - ℓ. Primary network input lines are labeled with the variable name. (Note that a, f, b, and i are defined as network lines, not primary network input lines, as they are branch lines from a fanout point on a primary network input line.) If a s-@-1 fault exists on a network line, say e, it is designated on the network diagram, Figure 2.3, with a small x on that line. The fault is denoted [17] as f_e^1; that is, line e stuck-at-1. A s-@-0 line fault is designated on the network diagram by a small o on the affected line.

<u>Definition 2.7</u>: A <u>path</u> is an ordered sequence of lines $\omega_1\omega_2\ldots\omega_r$ such that lines ω_i and ω_{i+1}, $1 \leq i \leq r-1$ are connected through either a single gate or a fanout point.

Thus, if we consider non-trivial paths, only ω_1 can be a primary network input line and only ω_r can be a network output. In Figure 2.3 $x_1acegj\ell$, de, k, and ehkℓ are all paths. The path k is termed a trivial path. In the case



FP-5047

Figure 2.3. Example alternating circuit.

of path $x_1acegj\ell$, network lines e and g, and x_1 and a, are connected through fanout points.

Definition 2.8: An output path is a path $\omega_1\omega_2\ldots\omega_r$ such that ω_r is the network output line.

The network in Figure 2.3 has many output paths. Examples are path $egj\ell$, $x_3dehk\ell$, and $dehk\ell$.

Definition 2.9: The inversion parity of a path, P, is the parity of the number of inversions (NAND gates, NOR gates, or inverters) encountered in tracing the specified path, P, through the circuit and is denoted by $\omega(P)$.

For the output paths noted above $\omega(egj\ell) = 0$, $\omega(x_3dehk\ell) = 0$, and $\omega(dehk\ell) = 1$.

Definition 2.10: A path $\omega_1\omega_2\ldots\omega_r$ is sensitized if for an input such that $d_1, d_2, \ldots, d_r$, $d_i \in \{0,1\}$, $1 \leq i \leq r$ is present on lines $\omega_1, \omega_2, \ldots, \omega_r$ the fault $f_{\omega_1}^{\bar{d}_1}$ forces $\omega_2, \ldots, \omega_r$ to take on values $\bar{d}_2, \bar{d}_3, \ldots, \bar{d}_r$.

Definition 2.11: A line fault f_a^d is sensitized if the network input is such that at least one path $\omega_1, \omega_2, \ldots, \omega_r$ is sensitized where $\omega_1 = a$ and ω_r is the network output, and the logical value of line a in the absence of the fault is $\bar{d}$.

For example, in Figure 2.3, the input $x_1x_2x_3 = 011$ sensitizes the fault f_e^o through the output path $ehk\ell$, since normally 1,1,0,1 is present on e,h,k,ℓ whereas the fault f_e^o forces h,k,ℓ to take on the values 0,1,0. Note that

$i = j = 1$ for $x_1x_2x_3 = 011$. The fault f_e^1 shown in Figure 2.3 is, however, not sensitized by 011 as the logical value on line e in the absence of the fault under 011 is 1, not 0. The input $x_1x_2x_3 = 010$ however sensitizes both f_e^1 and the path ehkℓ.

Now, with the foregoing definitions as a basis for terminology, consider again the irredundant alternating network N with n inputs $(x_1,\bar{x}_1)$, $(x_2,\bar{x}_2),\ldots,(x_n,\bar{x}_n)$ and one output y which represents a function f. Since N is irredundant and since all inputs $(X,\bar{X})$ are possible, to show that N is self-checking for an arbitrary single fault it is only necessary to show that for no input $(X,\bar{X})$ is an erroneous alternating output produced by the network due to the fault.

Lemma 2.1: An erroneous alternating output is produced by N under the presence of a single fault with input $(X,\bar{X})$ iff both X and $\bar{X}$ sensitize the fault.

Proof: Certainly if neither X nor $\bar{X}$ sensitize the fault, then the output of N under $(X,\bar{X})$ is the correct alternating output $y = (f(X),f(\bar{X})) = (f(X),\bar{f}(X))$. If only X sensitizes the fault then $y = (\bar{f}(X),f(\bar{X})) = (\bar{f}(X),\bar{f}(X))$ and the fault is detected. If only $\bar{X}$ sensitizes the fault then $y = (f(X),\bar{f}(\bar{X})) = (f(X),f(X))$ and the fault is detected. If both X and $\bar{X}$ sensitize the fault then $y = (\bar{f}(X),\bar{f}(\bar{X})) = (\bar{f}(X),f(X))$ and an undetected erroneous output is produced. Q.E.D.

Theorem 2.3: **Any arbitrary alternating network N is self-checking for single faults on primary network input lines.**

<u>Proof</u>: Assume primary input line x_i is stuck-at-d. If X sensitizes the fault, $x_i = \bar{d}$. But then $\bar{x}_i = d$, and $\bar{X}$ cannot sensitize the fault. Similarly, if $\bar{X}$ sensitizes the fault then $\bar{x}_i = \bar{d}$. But then $x_i = d$, and X cannot sensitize the fault. So, by Lemma 2.1, N is self-checking for all single faults on primary network input lines. Q.E.D.

So, having obtained Theorem 2.3, to show that an alternating network is self-checking for all single faults on primary network inputs and network lines it is only necessary to show that the network is self-checking for faults on network lines. Achieving the self-checking property for these faults, however, places some constraints on circuit structure.

2.4. Self-Checking Combinational Structures

Yamamoto, Watanabe, and Urano [36] have investigated standard two-level AND/OR and OR/AND alternating networks allowing inverters as a third level as required to obtain complemented variables. Such networks were shown to be self-checking for all single stuck-at faults on gate or inverter output lines. Other more general combinational network structures, however, can be shown to possess the same self-checking property under a generalized fault set. That fault set is the set of all single line faults on gate or inverter input or output lines. Three network structures will be considered, the internal fanout-free network, the essentially inverter-free AND/OR/NOT network, and the inverter-free network. First, consider the internal fanout-free network.

Definition 2.12: A network is internal fanout-free if each gate or inverter output is an input to at most one gate.

Theorem 2.4: Any internal fanout-free alternating network, N, is self-checking for single faults.

Proof: Let N be an internal fanout-free network containing a single fault on an arbitrary network line 'a', say f_a^d, where $d \in \{0,1\}$. By way of contradiction, assume that N is not self-checking for the fault f_a^d. Then there exists an input $(X,\overline{X})$ for which both X and $\overline{X}$ sensitize f_a^d producing an erroneous alternating output. That is, under X

$$y = d \oplus \omega(P_X)$$

where P_X is the output path from 'a' sensitized by X, and under $\overline{X}$

$$y = d \oplus \omega(P_{\overline{X}})$$

where $P_{\overline{X}}$ is the output path from 'a' sensitized by $\overline{X}$. But since N is internal fanout-free there is only one output path from 'a'. That is, $P_X = P_{\overline{X}} = P$ and the nonalternating output

$$(d \oplus \omega(P), \ d \oplus \omega(P))$$

is produced, a contradiction. Q.E.D.

An alternating internal fanout-free network is shown in Figure 2.4 a). The network is distinguished from a tree network in the strictest sense of the term since fanout of primary network inputs is allowed. The tree network is thus a subclass of the class of internal fanout-free networks and is self-checking for all single faults also.

Now, consider the essentially inverter-free AND/OR/NOT network.

Definition 2.13: An essentially inverter-free AND/OR/NOT network is an AND/OR/NOT network in which the input to any inverter is a network input.

Theorem 2.5: Any essentially inverter-free AND/OR/NOT network, N, is self-checking for all single faults.

Proof: The proof will be in two parts a) and b).

a) Consider a fault, f_a^d, where 'a' is any network line which is a gate input or output or an inverter output. If N is not self-checking for these faults then there is an input $(X,\overline{X})$ for which both X and $\overline{X}$ sensitize the fault producing an erroneous alternating output. Letting P_X and $P_{\overline{X}}$ denote the corresponding paths the output produced is

$$(d \oplus \omega(P_X), \quad d \oplus \omega(P_{\overline{X}})).$$

But, since $\omega(P_X) = \omega(P_{\overline{X}}) = 0$, the output produced becomes (d,d), which is not an erroneous alternating output.

b) Now, consider a fault, f_a^d, where 'a' is any inverter input line which is not a primary network input line. (Such a case exists when both an input and its complement are used to realize a function.) If N is not self-checking for these faults then there is an input $(X,\overline{X})$ for which both X and $\overline{X}$ sensitize the fault producing an erroneous alternating output. Letting P_X and $P_{\overline{X}}$ denote the corresponding paths the output produced is

$$(d \oplus \omega(P_X), \quad d \oplus \omega(P_{\overline{X}})).$$

But, since $\omega(P_X) = \omega(P_{\overline{X}}) = 1$, the output produced becomes $(\overline{d},\overline{d})$ which is not an erroneous alternating output. Q.E.D.

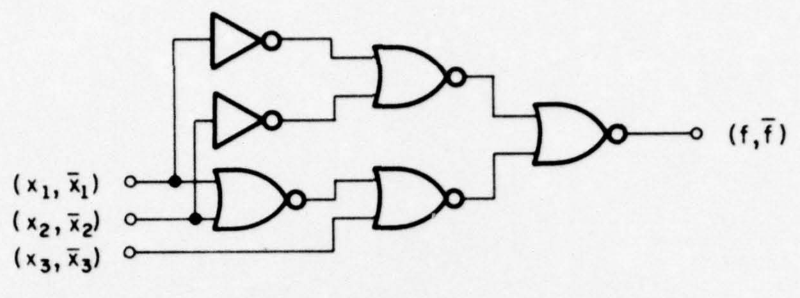
The alternating essentially inverter-free AND/OR/NOT network is a generalization of the structure studied by Yamamoto [36] in that an unlimited number of levels may be utilized in the AND/OR portion of the structure. An alternating network meeting the structural requirements is shown in Figure 2.4 b). Note that line a, the inverter input labeled in the figure is the type of network line considered in the b) part of the proof of Theorem 2.5.

The last specific structure to be examined is the inverter-free combinational structure.

Definition 2.14: A network is inverter-free if no inverter or inverting gate is present in the network structure.

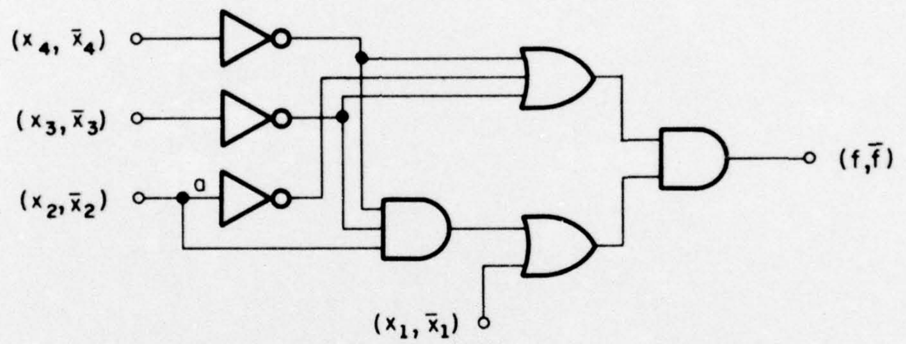
Theorem 2.6: Any inverter-free alternating network, N, is self-checking for all unidirectional faults.

Proof: By way of contradiction assume that N is not self-checking. Then for at least one input X and some unidirectional fault $f^d_{\omega_1} f^d_{\omega_2} \ldots f^d_{\omega_r}$ where $r \leq k$, the number of network lines, both X and $\overline{X}$ sensitize an element of $(f^d_{\omega_1}, f^d_{\omega_2}, \ldots, f^d_{\omega_r})$. Let X sensitize $f^d_{\omega_i}$, $\overline{X}$ sensitize $f^d_{\omega_j}$, where $1 \leq i,j \leq r$. Now, under $(X,\overline{X})$, $(\overline{f}(X),\overline{f}(\overline{X}))$ is produced since X and $\overline{X}$ both sensitize an element of the fault set. Further, under X, $y = d \oplus 0 = d$ is produced since all paths are of even parity. Similarly, under $\overline{X}$, $y = d \oplus 0 = d$ must be produced. But this implies that $\overline{f}(X) = \overline{f}(\overline{X}) \Rightarrow f(X) = f(\overline{X})$, a contradiction. So N is self-checking for all unidirectional faults. Q.E.D.



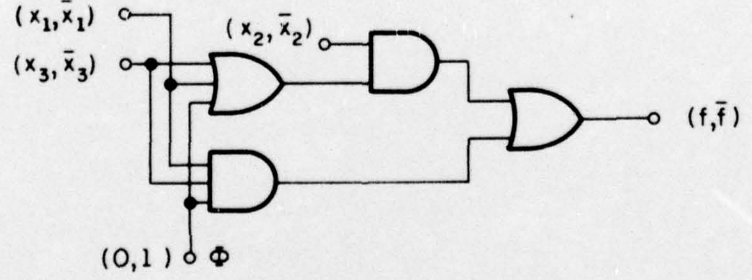
$f = \bar{x}_1 \bar{x}_2 + \bar{x}_1 x_3 + \bar{x}_2 x_3$

a) Internal Fanout-Free Network



$f = x_1 x_2 + x_1 \bar{x}_4 + x_2 \bar{x}_3 \bar{x}_4 + x_1 \bar{x}_3$

b) Essentially Inverter-Free AND/OR/NOT Network



$f = x_2(x_1 + x_3)$

c) Inverter Free Network

FP-5048

Figure 2.4. Self-checking alternating combinational structures.

An inverter-free alternating network is shown in Figure 2.4 c). The network represents the function $f = x_2(x_1+x_3)$. The dualization procedure outlined in Section 2.1 was used to obtain the network from the functional specification of f, thus requiring a clock Φ. A clock was not required for a) or b) as the function realized in each case was self-dual.

2.5. Necessary and Sufficient Self-Checking Conditions

Many combinational structures do not belong to one of the network structure classifications shown to be self-checking in Section 2.4. This does not mean that the network in question does not possess the self-checking property. Verification that the network is self-checking, however, is more difficult and may require computer simulation.

Theorem 2.7: If an alternating network N is self-checking for all faults on gate and inverter outputs, then it is self-checking for all single faults.

Proof: Let G be any network gate or inverter. Then

$$\omega(P_X) = \omega(P_{\overline{X}})$$

for any pair of output paths P_X and $P_{\overline{X}}$ sensitized by X and $\overline{X}$ respectively from the output of G. Then, any pair of output paths P'_X and $P'_{\overline{X}}$ sensitized by X and $\overline{X}$ respectively from an input to G, say a, can be written as

$$P'_X = a\ P_X \text{ for the path sensitized by } X \text{, and}$$
$$P'_{\overline{X}} = a\ P_{\overline{X}} \text{ for the path sensitized by } \overline{X}.$$

But then $\omega(P'_X) = \omega(P'_{\overline{X}})$ since $\omega(P_X) = \omega(P_{\overline{X}})$, and N must therefore be self-checking for a fault on 'a'. Therefore N is self-checking for all single faults. Q.E.D.

Thus, by Theorem 2.7, to show that a network is self-checking for all single faults, only faults on gate or inverter outputs need be examined. However, this constraint can be relaxed somewhat.

Definition 2.15: A fanout gate or fanout inverter is a gate or inverter whose output is connected to more than one gate or inverter.

Definition 2.16: A primary inverter is an inverter whose input is a primary network input line or a branch line from a primary network input fanout point. A secondary inverter is any inverter which is not a primary inverter.

Theorem 2.8: An alternating network N is self-checking for all single faults iff it is self-checking for all single faults on fanout gates and secondary fanout inverters.

Proof: A fault on any primary inverter input or output cannot be sensitized by both X and $\overline{X}$ and, therefore, cannot cause an erroneous alternating output. Thus, fanout from a primary inverter cannot affect the self-checking properties of N in any case. So, consider a fault on any other network line 'a' that is a fanout gate or secondary fanout inverter output line. Either there is a unique output path from 'a' or there is a unique path from 'a' to an input of a fanout gate or secondary fanout inverter. For the former case we know the network is self-checking for the fault by Theorem 2.4. So, consider the latter case. Any output path, P'_X, sensitized by X from 'a' can be written

$$P'_X = P^* P_X,$$

and any output path, $P'_{\overline{X}}$, sensitized by $\overline{X}$ from 'a' can be written

$$P'_{\overline{X}} = P^* P_{\overline{X}}$$

where P^* is the path from 'a' to the fanout gate or secondary fanout inverter input, and P_X and $P_{\overline{X}}$ are the output paths sensitized by X and $\overline{X}$ respectively from the output of the fanout gate or secondary fanout inverter. But,

$$\omega(P_X) = \omega(P_{\overline{X}}),$$

and therefore,

$$\omega(P'_X) = \omega(P'_{\overline{X}}).$$

So, N must be self-checking for a fault on 'a' in this case also. Q.E.D.

Theorem 2.9: An alternating network, N, is self-checking for all single faults iff the inversion parity of output paths originating on a given fanout gate or secondary fanout inverter output labeled 'a' sensitized by X equals the inversion parity of like output paths sensitized by $\overline{X}$ for every $(X,\overline{X})$ such that both X and $\overline{X}$ sensitize f_a^d for $d = 0,1$ and for every fanout gate or secondary fanout inverter.

Proof: By Theorem 2.8 the network N is self-checking for all single faults iff it is self-checking for all single faults on fanout gate and secondary fanout inverter outputs. So, consider an arbitrary fanout gate or secondary inverter, G, whose output, 'a', is stuck-at-d. If an erroneous output is to be produced as a result of the fault, by Lemma 2.1 both X and $\overline{X}$ must consider the latter case. Any output path, P'_X, sensitized by X from 'a' can be written

$$P'_X = P^* P_X,$$

and any output path, $P'_{\overline{X}}$, sensitized by $\overline{X}$ from 'a' can be written

$$P'_{\overline{X}} = P^* P_{\overline{X}}$$

where P^* is the path from 'a' to the fanout gate or secondary fanout inverter input, and P_X and $P_{\overline{X}}$ are the output paths sensitized by X and $\overline{X}$ respectively from the output of the fanout gate or secondary fanout inverter. But,

$$\omega(P_X) = \omega(P_{\overline{X}}),$$

and therefore,

$$\omega(P'_X) = \omega(P'_{\overline{X}}).$$

So, N must be self-checking for a fault on 'a' in this case also. Q.E.D.

Theorem 2.9: An alternating network, N, is self-checking for all single faults iff the inversion parity of output paths originating on a given fanout gate or secondary fanout inverter output labeled 'a' sensitized by X equals the inversion parity of like output paths sensitized by $\overline{X}$ for every $(X,\overline{X})$ such that both X and $\overline{X}$ sensitize f_a^d for $d = 0,1$ and for every fanout gate or secondary fanout inverter.

Proof: By Theorem 2.8 the network N is self-checking for all single faults iff it is self-checking for all single faults on fanout gate and secondary fanout inverter outputs. So, consider an arbitrary fanout gate or secondary inverter, G, whose output, 'a', is stuck-at-d. If an erroneous output is to be produced as a result of the fault, by Lemma 2.1 both X and $\overline{X}$ must sensitize the fault. But by hypothesis for any such $(X,\overline{X})$, $\omega(P_X) = \omega(P_{\overline{X}})$ and the detectable nonalternating output

$$(d \oplus \omega(P_X),\ d \oplus \omega(P_{\overline{X}}))$$

is produced. The network, N, must therefore be self-checking for the fault. Conversely, if there is an $(X,\overline{X})$ for which both X and $\overline{X}$ sensitize f_a^d for an arbitrary gate G with output 'a', while the inversion parity $\omega(P_X)$ does not equal the inversion parity $\omega(P_{\overline{X}})$, then that same $(X,\overline{X})$ produces on the output of N

$$(d \oplus \omega(P_X),\ d \oplus \omega(P_{\overline{X}}))$$

which in this case is an erroneous alternating output since $\omega(P_X) \neq \omega(P_{\overline{X}})$.

Q.E.D.

The significance of Theorems 2.7, 2.8, and 2.9 lies in the fact that in order to assure that a functional realization, which does not admit to a classification as one of the structures in Section 2.4, is still a self-checking realization, it is only necessary to simulate the output under fault conditions on fanout gate and secondary fanout inverter outputs and to compare these with the fault-free outputs so as to determine if an erroneous alternating output exists. Since such a simulation can be done quickly, an optimal minimization program such as Davidson's branch and bound program [12] could be modified so as to produce an optimal self-checking realization from a dualized functional specification, without degrading program performance significantly.

For some network structures not of the type considered in Section 2.4 the self-checking property can be deduced by inspection, and a computer simulation is therefore not required. Corollary 2.1 specifies the conditions.

Corollary 2.1: An alternating network, N, is self-checking for all single faults if all output paths from a given fanout gate or secondary fanout inverter are of identical parity for every fanout gate or secondary fanout inverter.

As an example, consider the alternating network shown in Figure 2.3. We know that the network is self-checking if it is self-checking for f_e^d, $d \in \{0,1\}$. However, only two output paths $egj\ell$ and $ehk\ell$ originate from the output of the only fanout gate or secondary fanout inverter, and they are both of the same inversion parity, $\omega(egj\ell) = \omega(ehk\ell) = 0$. So by Corollary 2.1 the network is self-checking.

Thus far only single output combinational networks have been considered. Multi-output networks are, however, important structures. If the network N is a multi-output network then the definition of self-checking is modified somewhat.

Definition 2.17: A multi-output network is self-checking for all single faults iff there exists no input $(X,\overline{X})$ for which an erroneous alternating output is produced unaccompanied by a nonalternating output on another network output line.

This definition is somewhat more relaxed than the definition in the single output case and should lead to simpler realizations when logic is shared in producing the required outputs than when each output is produced with independent logic. As an example of the multi-output alternating network consider the full adder.

Example 2.2: The full adder is shown in Figure 2.5 a) and Karnaugh maps specifying the outputs S_i and C_i as a function of the inputs C_{i-1}, a_i, and b_i are shown in Figure 2.5 b). From the map for S_i

$$S_i = C_{i-1}\bar{a}_i\bar{b}_i + \bar{C}_{i-1}\bar{a}_i b_i + C_{i-1}a_i b_i + \bar{C}_{i-1}a_i\bar{b}_i$$

$$\Rightarrow S_i = \bar{C}_{i-1}(\overline{\bar{a}_i\bar{b}_i + a_i b_i}) + C_{i-1}(\bar{a}_i\bar{b}_i + a_i b_i).$$

Similarly, from the map for C_i

$$C_i = a_i b_i + C_{i-1}b_i + C_{i-1}a_i$$

$$\Rightarrow C_i = C_{i-1}(\overline{\bar{a}_i\bar{b}_i}) + a_i b_i.$$

Based on these equations, a realization for the full adder is shown in Figure 2.5 c). The network must be an alternating network since C_i and S_i are self-dual functions. The network, however, is not self-checking. (The fault f_a^o produces an erroneous alternating output under $(C_{i-1}a_i b_i, \bar{C}_{i-1}\bar{a}_i\bar{b}_i) = (000,111)$.) A self-checking realization can be obtained by converting the structure shown in Figure 2.5 c) into an internal fanout-free structure. The resulting network is guaranteed to be self-checking and is shown in Figure 2.5 d).

Now consider those optimal full adder networks obtained by Liu, et al [22]. Of the thirty networks obtained, six presuppose the existence of only the input variable and not its complement. These six networks are of interest here and are shown in Figure 2.6 a)-f). Networks a) and b) are

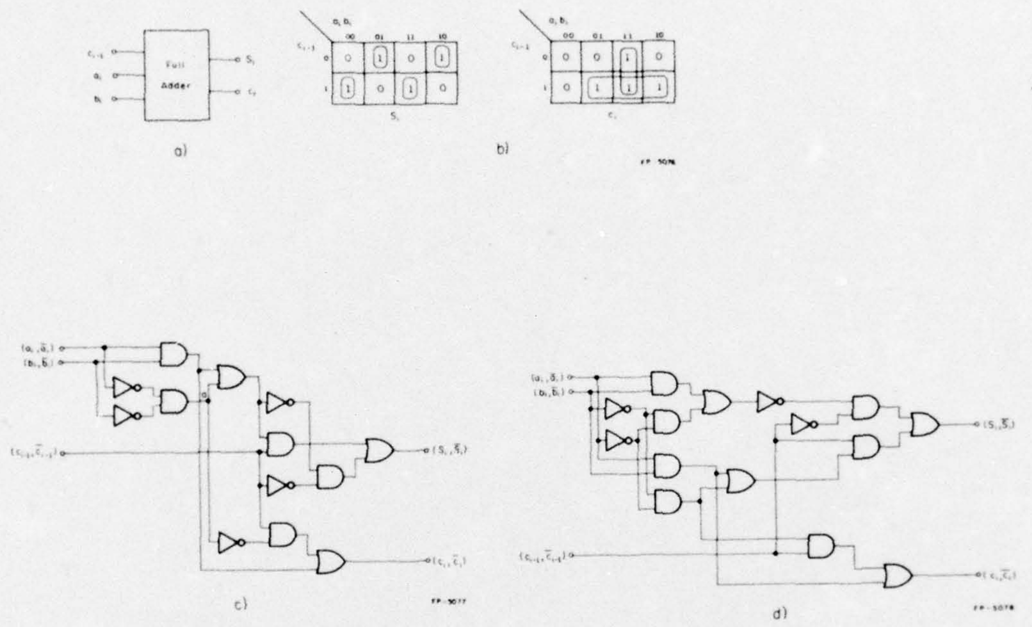


Figure 2.5. Alternating full adder.

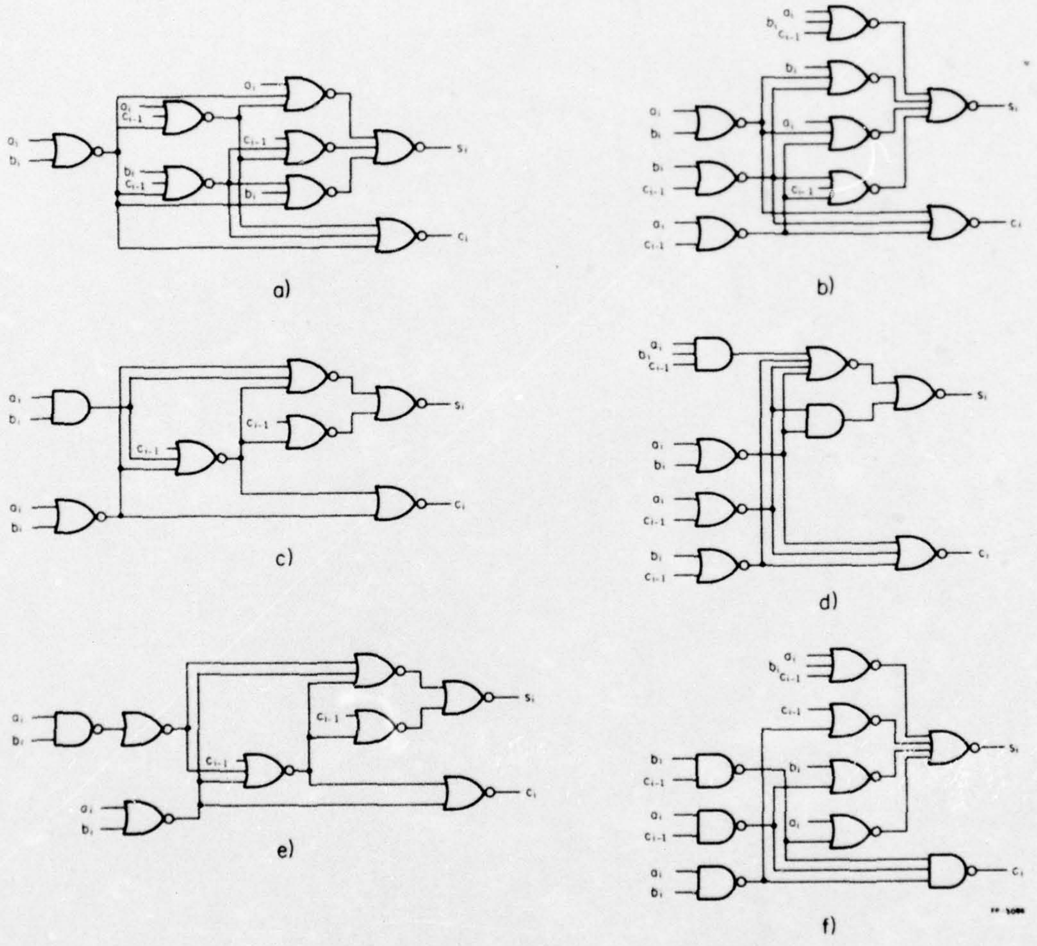


Figure 2.6. Optimal full adder.

optimal NOR adders; networks c) and d) are optimal NOR/AND adders; and networks e) and f) are optimal NOR/NAND adders. Adders a), c), and e) were optimized with no level restriction; adders b), d), and f) were limited to a maximum of 3 levels.

As to the self-checking properties of the optimal adders, adder b) meets the conditions of Corollary 2.1 and is therefore self-checking. Adder f) has no reconvergent fanout and is self-checking by Theorem 2.4 extended to the multi-output case. Adders a), c), d), and e) are at first glance not self-checking. But, computer simulations based on Theorems 2.8 and 2.9 verify the self-checking property of these networks also. For the adder in Figure 2.5 d) and the optimal adders b) and f) in Figure 2.6 no erroneous alternating output is ever produced on C_i or S_i. As a result the subnetwork realizing C_i and the subnetwork realizing S_i are therefore self-checking in themselves. For the optimal adders a), c), d), and e) in Figure 2.6, an erroneous alternating output can be produced on C_i or S_i, but such an output is always accompanied by a nonalternating output on S_i or C_i, respectively, thus meeting the requirements specified in Definition 2.17. But, in each case neither the subnetwork realizing C_i nor the subnetwork realizing S_i is self-checking in itself. Their union, however, sharing logic, is self-checking. This illustrates the advantages mentioned about sharing logic in the multi-output case.

3. SELF-CHECKING ALTERNATING SEQUENTIAL MACHINE

3.1. Sequential Machine Model

Consider the strongly connected synchronous sequential machine, $M(I,O,S,\delta,\lambda)$, with clock Φ of frequency f and period T represented schematically in Figure 3.1. The circuit has a finite number n of input terminals. The signals entering the circuit via these terminals constitute the set $\{x_1,x_2,\ldots,x_n\}$ of input variables. The set L of 2^n distinct inputs is called the input alphabet I, and each configuration is referred to as a symbol of the alphabet. Thus the input alphabet is given by

$$I = \{I_o,I_1,\ldots,I_L\} \text{ for } I_p = x_1x_2\ldots x_n,\ 0\leq p\leq L \text{ and } x_i\epsilon\{0,1\},\ \ 1\leq i\leq n.$$

Similarly, the circuit has a finite number ℓ of output terminals which define th set $\{z_1,z_2,\ldots,z_\ell\}$ of output variables. The set M of 2^ℓ ordered ℓ-tuples is called the output alphabet and is given by

$$0 = \{0_o,0_1,\ldots,0_M\} \text{ for } 0_q = z_1z_2\ldots z_\ell,\ \ 0\leq q\leq M \text{ and } z_i\epsilon\{0,1\},\ 1\leq i\leq \ell.$$

The signal at the output of each memory element is referred to as a state variable, and the set $\{y_1,y_2,\ldots,y_m\}$ constitutes the set of state variables. The combination of values at the outputs of the m memory elements $y_1,y_2,\ldots,y_m$ defines the present state of the machine. The set R of 2^m m-tuples constitute the entire set of states of the machine

$$S = \{S_o,S_1,\ldots,S_R\} \text{ for } S_s = y_1y_2\ldots y_m,\ \ 0\leq s\leq R \text{ and } y_i\epsilon\{0,1\},\ 1\leq i\leq m.$$

The external inputs $x_1,x_2,\ldots,x_n$ and the values of the state variables $y_1,y_2,\ldots,y_m$ are applied to the combinational circuit, C. This circuit in turn produces the outputs $z_1,z_2,\ldots,z_\ell$ and the feedback variables

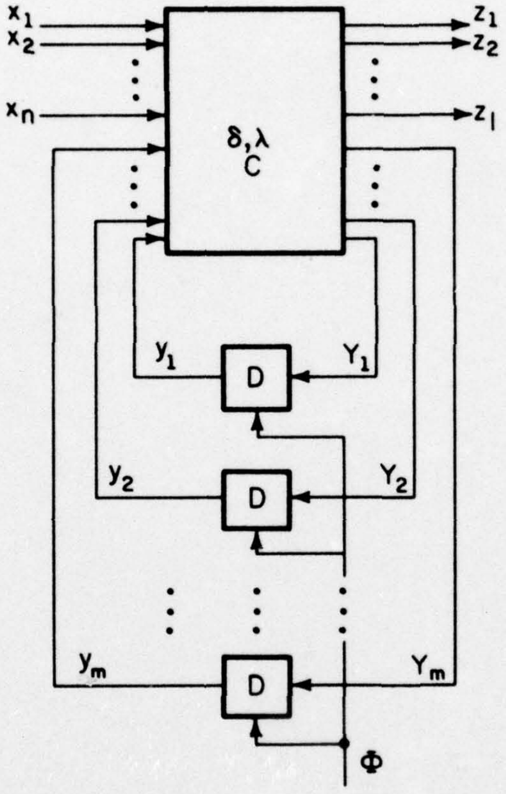


Figure 3.1. Synchronous machine, M.

$Y_1, Y_2, \ldots, Y_m$. The values of the Y's which appear at the output of C at time t are identical with the values of the state variables at t+T and therefore define the next state of the machine. The machine M thus realizes the transformations

$$\delta: \; S \times I \rightarrow S$$

$$\lambda: \; S \times I \rightarrow O.$$

Normally these transformations are described by means of a state table or a state diagram. Of interest here is the state table. It has $a = 2^n$ columns, one for each input symbol, and b rows, one for each state of the machine. For each combination of input symbol and present state, the corresponding entry specifies the output that will be generated and the next state to which the machine will go. The succession of states through which a sequential machine passes, and the output sequence which it produces in response to a known input sequence, are specified uniquely by the state table and the initial state, where by initial state we refer to the state of the machine prior to the application of the input sequence.

3.2. Alternating Sequential Machine

An alternating sequential machine M_A associated with M can be defined. Such a machine has the property that state variables and output variables alternate in response to alternating inputs.

Definition 3.1: An alternating sequential machine associated with M is a sequential machine having the property that for inputs $(I_i, \overline{I}_i)$, $1 \leq i \leq L$, alternating in synchronism with Φ, secondary variables y_j, $1 \leq j \leq m$, and output variables z_k, $1 \leq k \leq \ell$, alternate in synchronism with Φ, while maintaining the next state and output relations realized by M.

Thus, for any alternating sequential machine associated with M,

$$I_A = \{(I_o,\overline{I}_o),(I_1,\overline{I}_1),\ldots,(I_L,\overline{I}_L)\}$$

$$O_A = \{(O_o,\overline{O}_o),(O_1,\overline{O}_1),\ldots,(O_M,\overline{O}_M)\}$$

$$S_A = \{(S_o,\overline{S}_o),(S_1,\overline{S}_1),\ldots,(S_R,\overline{S}_R)\}.$$

Now, consider the sequential machine, M^*, modeled in Figure 3.2. The inputs $x_1,x_2,\ldots,x_n$ are assumed to alternate in synchronism with Φ. The combinational circuitry, C_A, is a self-checking alternating circuit realizing the functions δ and λ of M. The clock line, Φ, of period T and frequency f shown as an input to C_A, is used as the added variable required to dualize δ and λ if they are not already self-dual. And, the clock, Φ_A, of period T/2 and frequency 2f, is in phase with Φ. (The clock Φ can be considered to be derived from Φ_A.) To analyze the operation of M^* assume that time t is the beginning of an arbitrary clock period and that at time $t, y_1y_2\ldots y_m = S_s$ and $y_1'y_2'\ldots y_m' = \overline{S}_s$ (Figure 3.3). If $(I_k,\overline{I}_k)$ is applied as input at time t then $I(t) = I_k$, $I(t+\frac{T}{2}) = \overline{I}_k$, since inputs alternate in synchronism with Φ. Further,

$$y_i'(t) = \overline{y}_i(t),$$

$$y_i(t+\tfrac{T}{2}) = \overline{y}_i(t).$$

Thus, all inputs to C_A and therefore all outputs from C_A alternate in synchronism with Φ. Since C_A realizes δ in producing Y_i, $1 \leq i \leq m$,

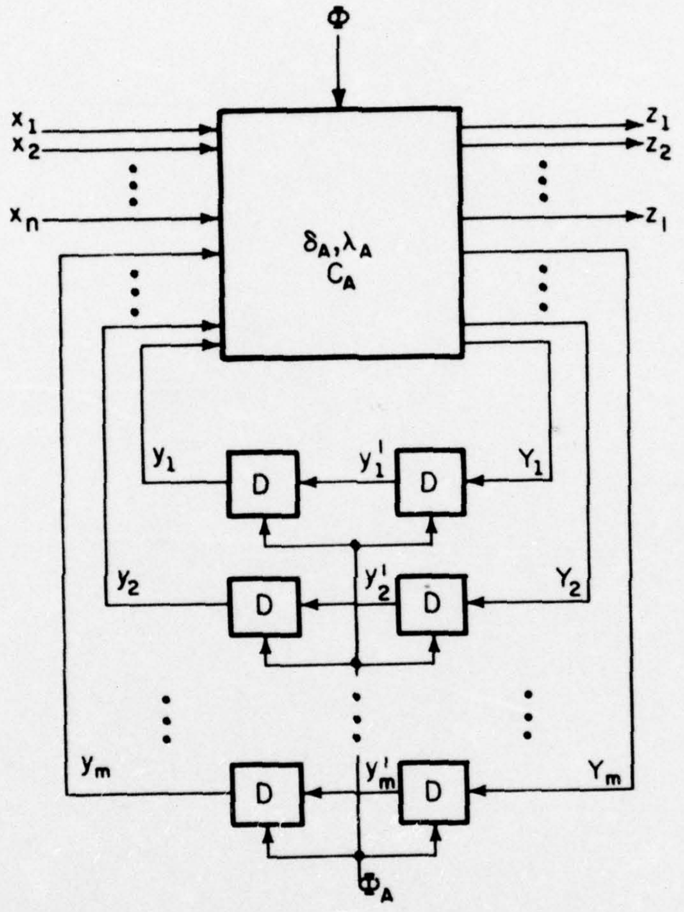


Figure 3.2. Synchronous machine, M^*.

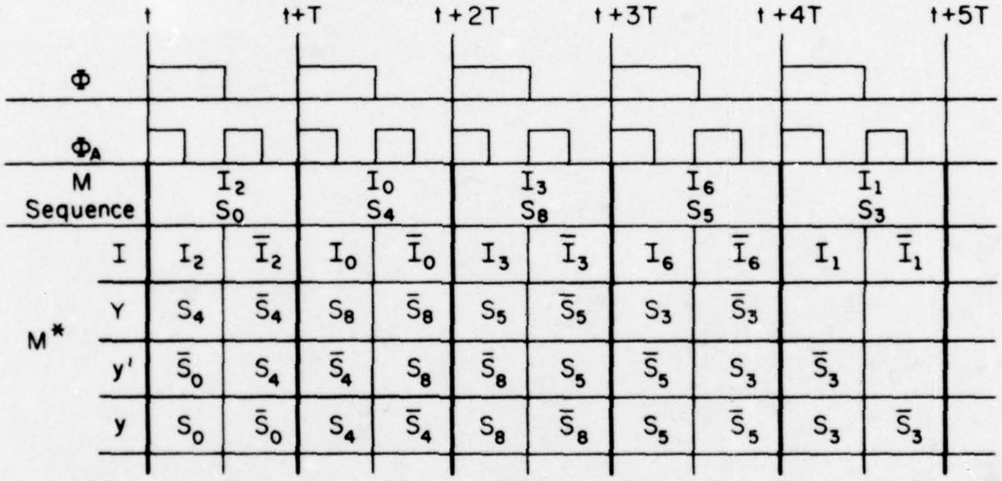

t
t+T
t+2T
t+3T
t+4T
t+5T
Φ
Φ_A
M Sequence
I_2 S_0
I_0 S_4
I_3 S_8
I_6 S_5
I_1 S_3
I
I_2 $\bar{I}_2$ I_0 $\bar{I}_0$ I_3 $\bar{I}_3$ I_6 $\bar{I}_6$ I_1 $\bar{I}_1$
M^*
Y
S_4 $\bar{S}_4$ S_8 $\bar{S}_8$ S_5 $\bar{S}_5$ S_3 $\bar{S}_3$
y'
$\bar{S}_0$ S_4 $\bar{S}_4$ S_8 $\bar{S}_8$ S_5 $\bar{S}_5$ S_3 $\bar{S}_3$
y
S_0 $\bar{S}_0$ S_4 $\bar{S}_4$ S_8 $\bar{S}_8$ S_5 $\bar{S}_5$ S_3 $\bar{S}_3$

Figure 3.3. Timing relationship.

$$Y_i(t) = \delta_A(I(t),S(t)) = \delta(I_k,S_s)$$

$$Y_i(t+\frac{T}{2}) = \delta_A(I(t+\frac{T}{2}),S(t+\frac{T}{2})) = \delta_A(\overline{I}_k,\overline{S}_s) = \overline{\delta}(I_k,S_s).$$

Therefore,

$$y_i(t+T) = y_i'(t+\frac{T}{2}) = Y_i(t) = \delta(I_k,S_s)$$

$$y_i(t+\frac{3T}{2}) = y_i'(t+T) = Y_i(t+\frac{T}{2}) = \overline{\delta}(I_k,S_s)$$

and since C_A realizes λ in producing z_i, $1 \leq i \leq \ell$,

$$z_i(t) = \lambda_A(I(t),S(t)) = \lambda(I_k,S_s)$$

$$z_i(t+\frac{T}{2}) = \lambda_A(I(t+\frac{T}{2}),S(t+\frac{T}{2})) = \lambda_A(\overline{I}_k,\overline{S}_s) = \overline{\lambda}(I_k,S_s).$$

The sequential machine M* thus produces Φ-synchronized alternating secondary variables and outputs in response to Φ-synchronized input variables while maintaining the next state and output relations realized by M as required by Definition 3.1, and is therefore an alternating sequential machine associated with M provided the initial state is set in the y flipflops and the complement of the initial state is set in the y' flipflops on initialization.

3.3. Self-Checking Capability

As could be expected, the alternating sequential machine, M*, has some fault detecting capabilities. In fact, all single stuck-at faults on lines in the feedback and output combinational circuitry, C_A, and on the memory element input and output lines can be detected. If, in addition, it can be guaranteed that no erroneous alternating outputs are produced prior to the fault detection then M^* is a self-checking alternating sequential machine. The ideas are formalized by the following definition.

Definition 3.2: The alternating sequential machine, M^*, is self-checking for all faults from a fault set F iff for every fault from F there exists a state $(S_i,\overline{S}_i)\epsilon S_A$ and an input $(I_j,\overline{I}_j)\epsilon I_A$ which produces a nonalternating (detectable) output on $Y_1,Y_2,\ldots,Y_m$, $z_1,z_2,\ldots,z_{\ell-1}$ or z_ℓ and no state $(S_i,\overline{S}_i)\epsilon S_A$ and input $(I_j,\overline{I}_j)\epsilon I_A$ for which an erroneous alternating output is produced prior to a nonalternating output.

Since, in general, the alternating circuit C_A in M^* is not exercised by all possible input combinations, it is assumed that C_A has been designed to be irredundant with respect to the input combinations encountered. With this as an assumption the following theorem can be proved.

Theorem 3.1: The alternating sequential machine, M^*, is self-checking for all single faults on lines in C_A or on input and output lines associated with the memory elements.

Proof: First consider a fault on any line in C_A. Since M is strongly connected so is M^*. So, any state in M^* can be reached from any other state by the application of the proper input sequence. And since C_A is self-checking and irredundant with respect to the state and input space there is a state $(S_i,\overline{S}_i)$ and an input $(I_j,\overline{I}_j)$ which produces a nonalternating output on some output of C_A, and no state or input which produces an erroneous alternating output prior to such a nonalternating output. Since the non-alternating output appears on $Y_1,Y_2,\ldots,Y_m$, $z_1,z_2,\ldots,z_{\ell-1}$ or z_ℓ, M^* must be self-checking for these faults. Now, consider a stuck-at-d fault on any memory input or output line. Such a fault must manifest itself eventually as y_i stuck-at-d for some i, $1 \leq i \leq m$. Since C_A is self-checking for all single faults on primary network inputs, a nonalternating output appears on $Y_1, Y_2, \ldots, Y_m$, $z_1, z_2, \ldots, z_{\ell-1}$ or z_ℓ simultaneously with or prior to an erroneous alternating output and M^* must be self-checking for these faults.

Q.E.D.

The self-checking capability of the sequential machine M^* was obtained by encoding the state variables y_i, $1 \leq i \leq m$, in time. While such an encoding has the advantage that the secondary state variable assignment is not constrained, it has the disadvantage of doubling the number of memory elements required. However, the added set of memory elements serve a double purpose as will be shown in the following chapter.

To illustrate the techniques involved in designing an alternating sequantial machine consider an alternating sequential machine representing a one-input, one-output sequence detector which produces an output 1 every time the sequence 0101 is detected and an output 0 at all other times.

Example 3.1: The 0101 sequence detector has been designed as a conventional sequential machine in Kohavi [21]. The state table for the machine is given in Figure 3.4 a). As there are four distinct states in the state table, two memory elements are required in the sequential machine. Selecting the state assignment

PS	NS, z_1	
	$x_1 = 0$	$x_1 = 1$
S_0	S_1,0	S_0,0
S_1	S_1,0	S_2,0
S_2	S_4,0	S_0,0
S_3	S_1,0	S_2,1

a) State Table

$y_1 y_2$	$Y_1 Y_2$		z_1	
	$x_1 = 0$	$x_1 = 1$	$x_1 = 0$	$x_1 = 1$
00	01	00	0	0
01	01	11	0	0
11	10	00	0	0
10	01	11	0	1

b) Transition and Output Tables

z_1

x_1 \ $y_1 y_2$	00	01	11	10
0	0	0	0	0
1	0	0	0	1

Y_1

x_1 \ $y_1 y_2$	00	01	11	10
0	0	0	1	0
1	0	1	0	1

Y_2

x_1 \ $y_1 y_2$	00	01	11	10
0	1	1	0	1
1	0	1	0	1

c) Excitation and Output Maps

FP-5099

Figure 3.4. Sequence detector.

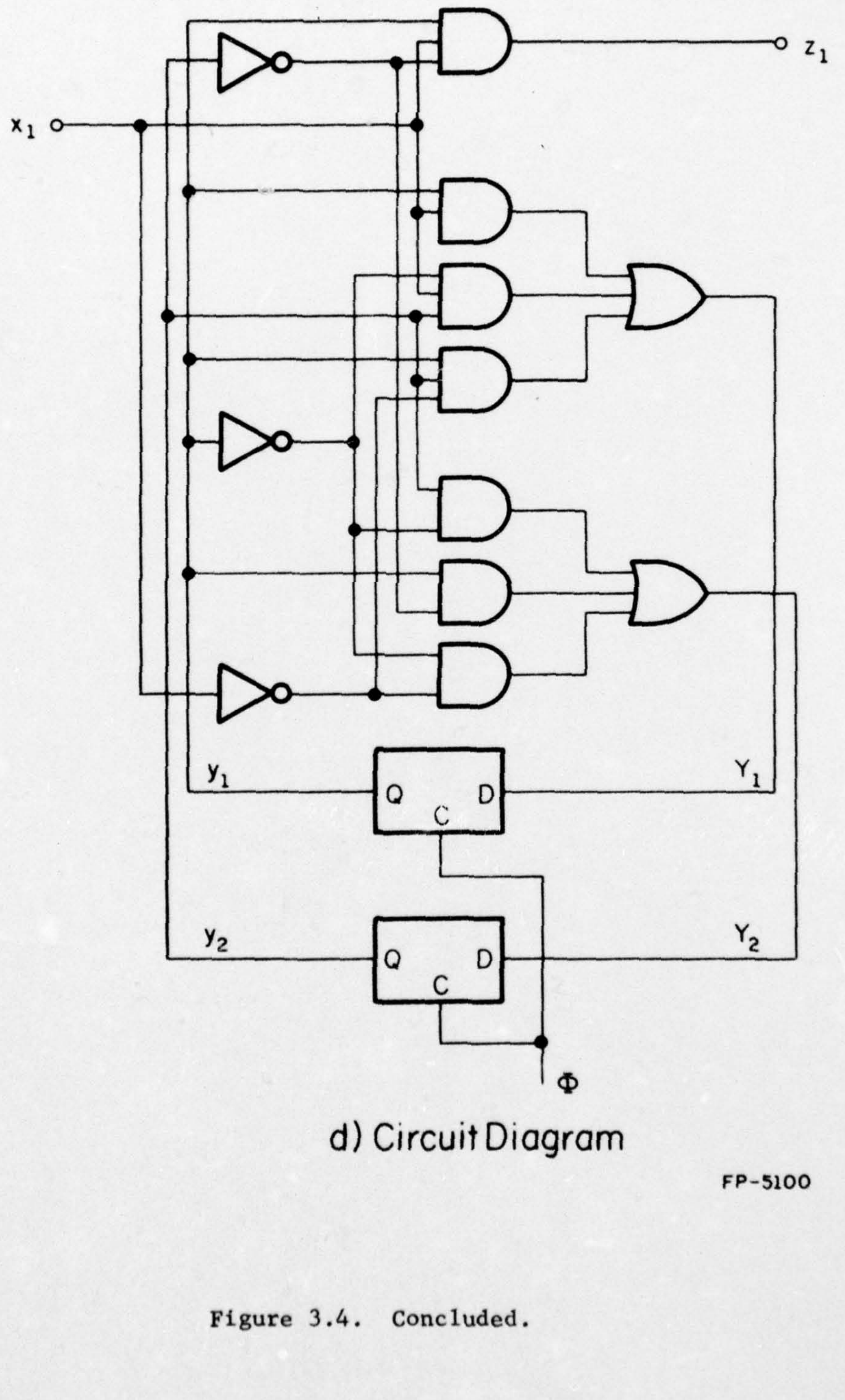


Figure 3.4. Concluded.

$$y_1y_2 = 00 = S_o$$
$$y_1y_2 = 01 = S_1$$
$$y_1y_2 = 11 = S_2$$
$$y_1y_2 = 10 = S_3,$$

transition and output tables can be constructed, Figure 3.4 b). From these tables Karnaugh maps specifying the output function, z_1, and the next-state variables, Y_1 and Y_2, can be constructed, Figure 3.4 c) and standard two-level realizations of z_1, Y_1, and Y_2 yield the conventional 0101 sequence detector, Figure 3.4 d).

The alternating sequential machine representing the 0101 sequence detector is obtained by first dualizing the functional descriptions of z_1, Y_1, and Y_2 to form functional specifications for z_{1*}, Y_{1*}, and Y_{2*}. These specifications are shown as Karnaugh maps in Figure 3.5. These functions are then realized in two-level logic and utilized as C_A in the alternating sequential machine, Figure 3.6. The combinational block C_A thus has as inputs $(x_1,\overline{x}_1)$ and the (0,1) clock Φ, and as outputs, the machine output $(z_1,\overline{z}_1)$ and the next-state outputs $(Y_1,\overline{Y}_1)$ and $(Y_2,\overline{Y}_2)$. Four D-flipflop memory elements are required in the alternating machine, and these are configured as specified in Figure 3.2, clocked by a clock Φ_A of period T/2. The conventional machine utilized 12 gates and 2 D-flipflop memory elements. The alternating machine required 19 gates and 4 D-flipflop memory elements. Thus, gate count was increased by roughly a factor of 1.5 and memory element

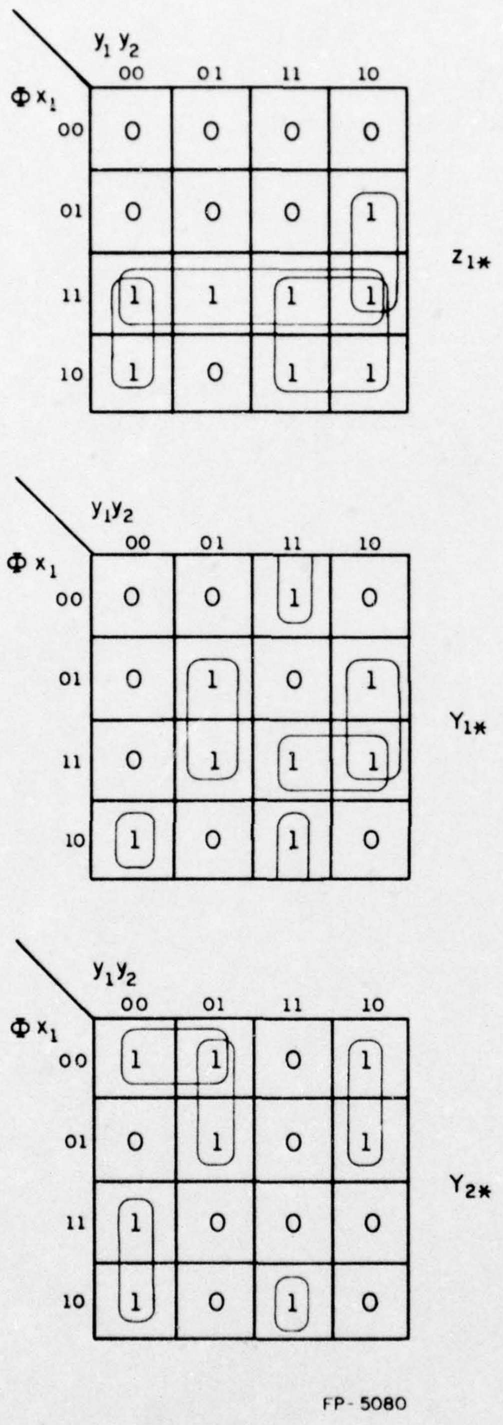
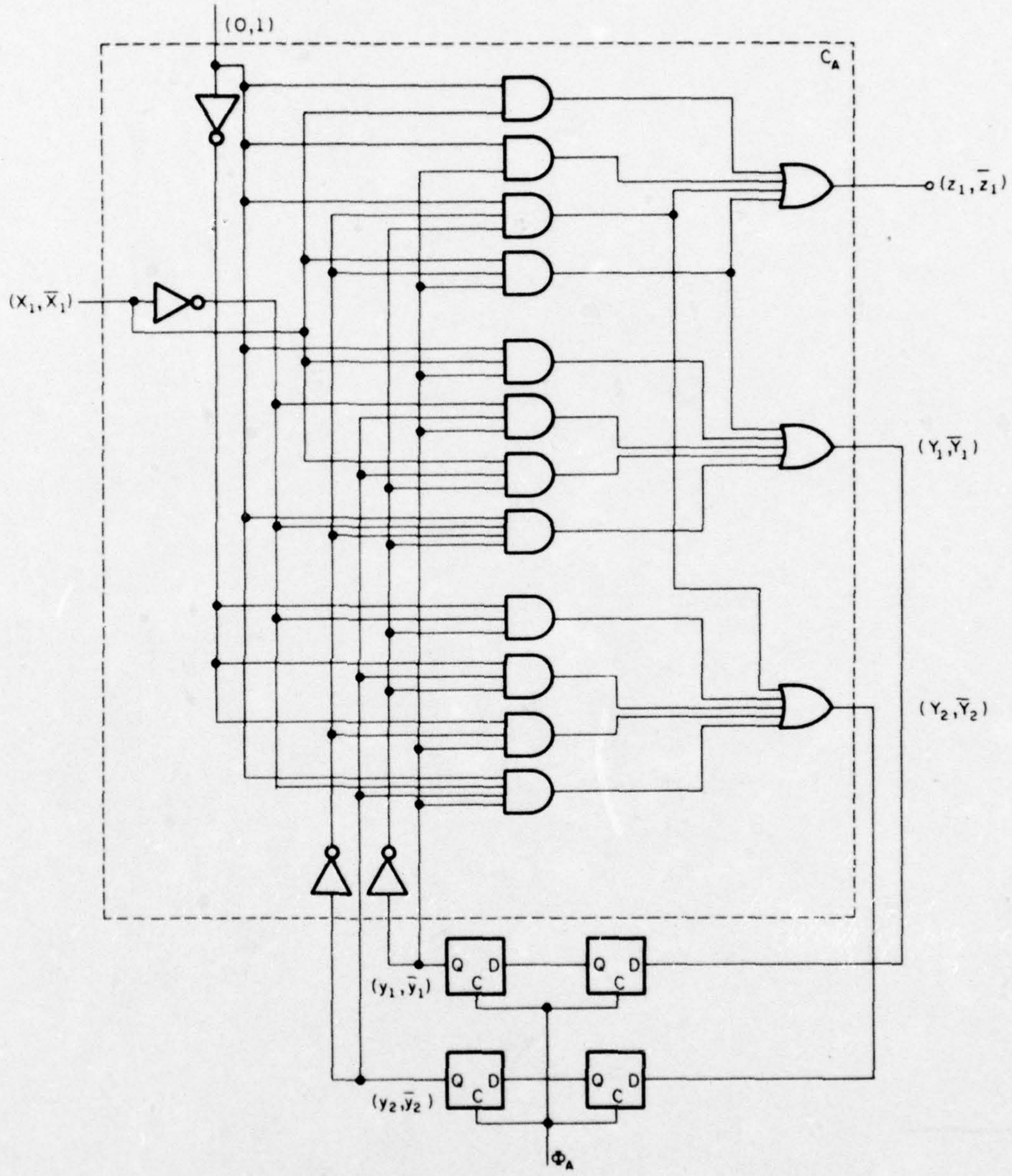


Figure 3.5. Dualized functional descriptions.



FP-5081

Figure 3.6. Alternating sequence detector.

count by 2. Also, to this added cost must be included the cost of the alternating checker which monitors $(z_1,\overline{z}_1)$, $(Y_1,\overline{Y}_1)$, and $(Y_2,\overline{Y}_2)$ so as to indicate the presence of nonalternating signals on these lines. The alternating checker is the subject of the next chapter.

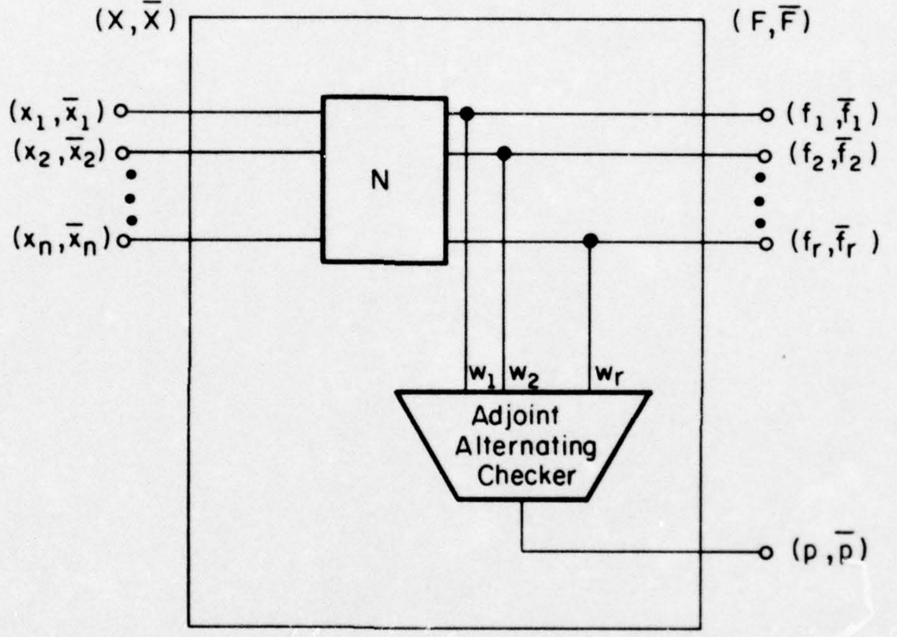
4. ALTERNATING CHECKERS

4.1. Adjoint Alternating Checker

In any practical alternating logic network, some means must be provided to monitor network output lines (in the case of synchronous machines, network output lines and feedback lines) so as to provide an indication of the occurrence of nonalternating signals on these lines. Any network which performs this function is an alternating checker. The adjoint alternating checker is one such checker.

Definition 4.1: An adjoint alternating checker is a single output network which produces in response to synchronized alternating inputs a synchronized alternating output and produces in response to one or more synchronized nonalternating inputs a synchronized nonalternating output.

An adjoint alternating checker monitoring an r output alternating combinational network N is shown in Figure 4.1. The outputs of the network N, $f_1,f_2,\ldots,f_r$ are system outputs but also are inputs to the alternating checker. The output of the alternating checker, p, is also a system output and is used for fault indication. Under fault-free conditions in response to a synchronized alternating input $(X,\overline{X})$, r synchronized alternating outputs $(f_1,\overline{f}_1),(f_2,\overline{f}_2),\ldots,$ and $(f_r,\overline{f}_r)$ are produced by N. These outputs are monitored by the checker which produces a synchronized alternating output $(p,\overline{p})$ in response to them. If a fault exists in N and N is self-checking there is an input $(X,\overline{X})$ for which a nonalternating output, (f_i,f_i) or $(\overline{f}_i,\overline{f}_i)$, is produced on some output f_i. This nonalternating output as



FP-5082

Figure 4.1. Alternating network N and adjoint checker.

input to the checker causes a nonalternating checker output, (p,p) or $(\overline{p},\overline{p})$, indicating the presence of that fault. A combinational realization for the adjoint alternating checker, however, does not exist as the following theorem shows.

Theorem 4.1: A combinational adjoint alternating checker does not exist.

Proof: Assume that the adjoint checker with r inputs $\omega_1,\omega_2,\omega_3,\ldots,$ and ω_r and one output p can be realized with combinational logic and suppose that $\omega_1\omega_2\ldots\omega_r = v_1v_2\ldots v_r$ maps to d where $v_1,v_2,\ldots,v_r$ and $d\epsilon\{0,1\}$. Then $\overline{v}_1\overline{v}_2\ldots\overline{v}_r$ must map to $\overline{d}$ as the output must alternate when alternating inputs are applied. Further, all other $\omega_1\omega_2\ldots\omega_r$ such that $\omega_1\omega_2\ldots\omega_r \neq v_1v_2\ldots v_r$ must map to d since the output must be nonalternating for nonalternating inputs. Now consider $\overline{v}_1v_2\ldots v_r$ and $v_1\overline{v}_2\overline{v}_3\ldots\overline{v}_r$. Both must map to d as neither equals $v_1v_2\ldots v_r$ and, therefore, under $(\overline{v}_1v_2\ldots v_r,v_1\overline{v}_2\overline{v}_3\ldots\overline{v}_r)$, an alternating input, the nonalternating output (d,d) is produced. The adjoint alternating checker cannot, therefore, be combinational. Q.E.D.

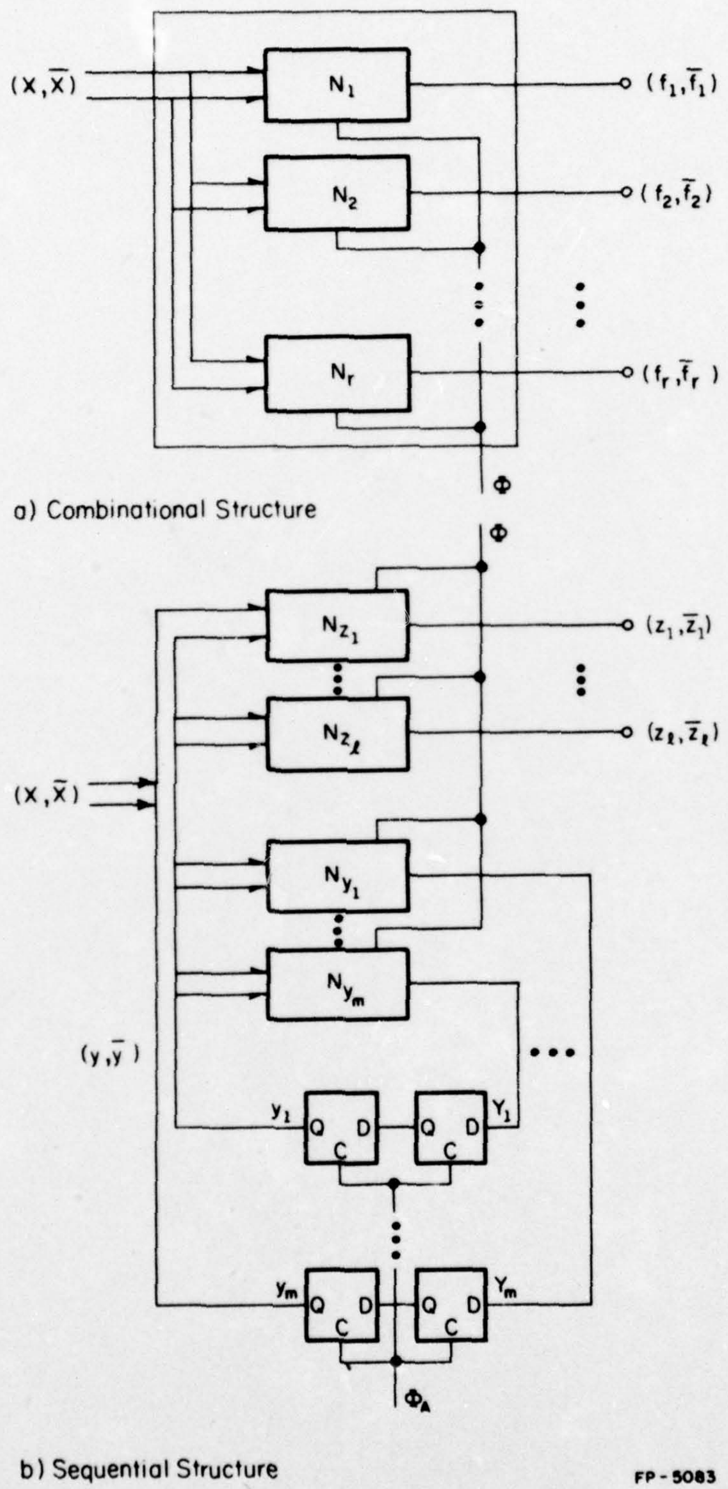
While the adjoint alternating checker cannot be realized in a combinational structure, a combinational alternating checker exists which is capable of performing the checking function for certain classes of network structures. This checker is discussed in the following section.

4.2. Semi-Adjoint Alternating Checker

Some alternating networks are structured such that as a result of a stuck-at fault, at most one network output (or feedback line in the case of the synchronous machine) is affected by the fault under any given input, where the effect of the fault must be the production of a non-alternating output on that line. A combinational network and a sequential network meeting this criterion are shown in Figure 4.2. In the combinational structure lines $f_1,f_2,\ldots,f_r$ are monitored lines. And, since a network fault in any subnetwork N_i, $1 \leq i \leq r$, can only affect the network output f_i, only one nonalternating output can be produced at any time as a result of that fault. Obviously, the result is achieved by eliminating shared logic in producing $f_1,f_2,\ldots,f_r$. In the sequential network, the same technique is used. In this case monitored lines are $y_1,y_2,\ldots,y_m$ and $z_1,z_2,\ldots,z_\ell$, namely all network output and feedback lines. Any single fault in network N_{z_i}, $1 \leq i \leq \ell$, affects only z_i. Any single fault in network N_{Y_i}, $1 \leq i \leq m$, affects only Y_i *and therefore* only y_i. Similarly, any line fault on a memory element input or output line affects only one monitored line, y_i. Any checker monitoring lines $z_1,z_2,\ldots,z_\ell,y_1,y_2,\ldots,y_\ell$ therefore receives as input at most one nonalternating signal at any time, while all others alternate. The same can be said for any checker monitoring lines $f_1,f_2,\ldots,$ and f_r in the combinational network.

The above structures relax some of the restrictions placed on the checker, allowing the use of a combinational alternating checker.

Definition 4.2: A semi-adjoint alternating checker is an r input, 1 output network which produces in response to r synchronized alternating inputs a synchronized alternating output and produces in response to r-1 alternating and 1 nonalternating synchronized inputs a nonalternating output.



FP-5083

Figure 4.2. Restricted combinational and sequential networks.

Now, consider an r-input exclusive-OR tree where r is odd, and let the inputs be designated $\omega_1, \omega_2, \ldots, \omega_r$.

Theorem 4.2: The r-input exclusive-OR tree, C, is a semi-adjoint alternating checker for odd r.

Proof: Assume that synchronized alternating inputs are applied, $(\omega, \overline{\omega}) = (\omega_1\omega_2\ldots\omega_r, \overline{\omega}_1\overline{\omega}_2\ldots\overline{\omega}_r)$ where ω has ℓ 1's and $\overline{\omega}$, m 1's. Further, assume that C produces (a,b) in response to input $(\omega, \overline{\omega})$. Now, since the exclusive-OR tree produces an output of 1 when the parity of the number of 1's on its input lines is odd, $a = P_o(\ell)$ where P_o is the odd parity function. Similarly, $b = P_o(m)$. But $m = r-\ell$ for alternating inputs. So, $b = P_o(r-\ell)$. Now if ℓ is odd $a = 1$. However, since r is odd, $r-\ell$ must be even, $P_o(r-\ell) = 0$, and $b = 0$. An alternating output is thus produced. If ℓ is even, $a = 0$, $r-\ell$ is odd, $P_o(r-\ell) = 1$, and $b = 1$. An alternating output is thus produced for this case also.

If at most one input does not alternate then in effect the circuit monitors the parity of the other alternating inputs producing an even parity function P_e if its nonalternating line is (1,1) and an odd parity function P_o if the nonalternating line is (0,0). Letting ℓ' be the number of 1's on the alternating lines in ω and m' be the number of 1's on the alternating lines in $\overline{\omega}$, $a = P_e(\ell')$, $b = P_e(r-1-\ell') = P_e(\ell') = a$ since r-1 is even if the nonalternating line is (1,1). And, $a = P_o(\ell)$, $b = P_o(r-1-\ell') = P_o(\ell') = a$ if the line is (0,0). Thus, a nonalternating output is produced. The exclusive-OR tree is therefore a semi-adjoint alternating checker for odd r.

Q.E.D.

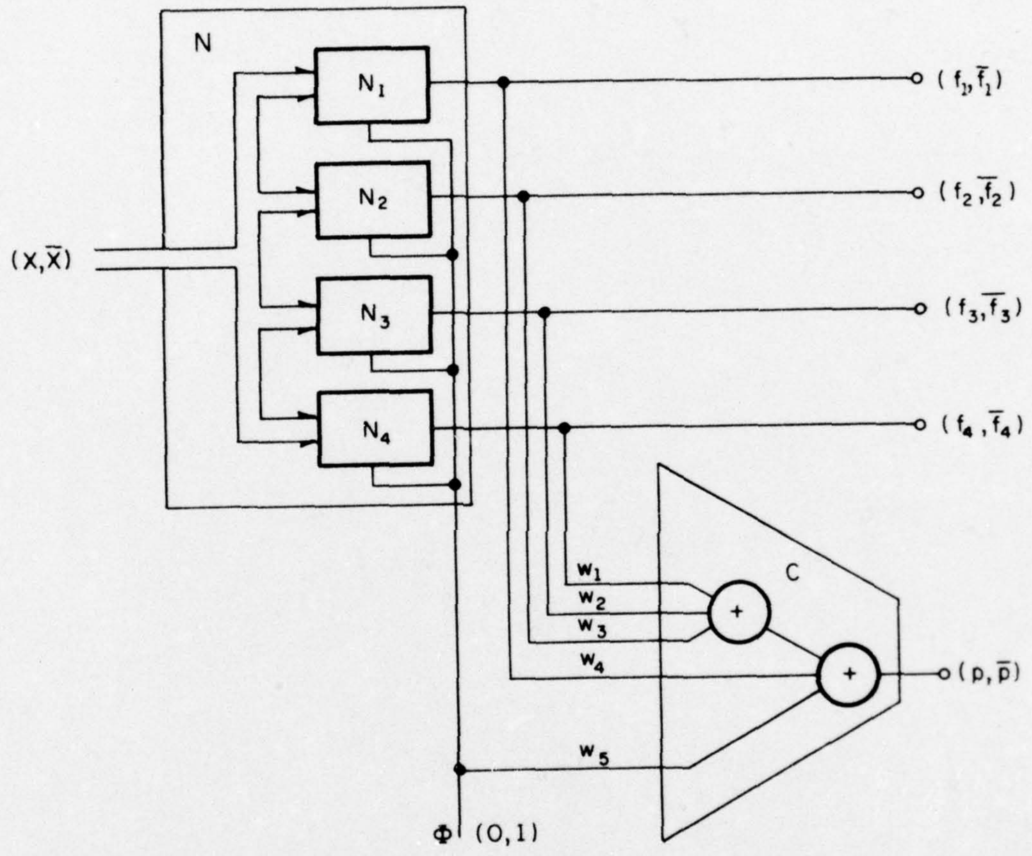
If an even number of lines is to be monitored it is only necessary to tie the unused checker input line to the clock Φ. This application is illustrated in Figure 4.3.where 4 output lines from a combinational network are monitored using a 5-input exclusive-OR tree composed of 3-input exclusive-OR gates. Note that the combinational network does not utilize shared logic and can therefore be monitored with a semi-adjoint alternating checker as shown.

So far, only faults in the monitored network have been considered. Faults, however, may also occur in the checker itself. The self-testing property of the alternating checker assures that a fault signal is produced on the checker output as a result of that fault while the monitored network functions normally.

Definition 4.3: An alternating checker is self-testing if for every single stuck-at fault on checker gate input and output lines, there exists a network output which as input to the checker produces a fault indication as output.

Theorem 4.3: The exclusive-OR semi-adjoint alternating checker is self-testing if the number of gate inputs to each exclusive-OR gate is odd.

Proof: Consider a fault on line g, f_g^d, in an exclusive-OR tree. The fault partitions the set of checker inputs $\{\omega_1,\omega_2,\ldots,\omega_r\}$ into two sets. One set, S, contains those ω_i which are inputs to the subtree whose output is g and the other set, R, contains all remaining inputs. Now, the number of inputs to the subtree whose output is g must be odd since only odd input exclusive-OR gates are used. That is, $\#|S| = a$ is odd. But then, $\#|R| = r-a = b$ where



FP - 5084

Figure 4.3. Multi-output combinational network and semi-adjoint alternating checker.

r is the total number of checker inputs. Since r is odd and a is odd, b is even. Now, consider an input ω such that $\omega_i = 1$ if $\omega_i \in R$. Since the faulty checker essentially performs the function $C_{f_g^d} = (\sum_{\omega_i \in R} \omega_i) \oplus d$, d is produced with ω as input. With $\overline{\omega}$ as input, every $\omega_i \in R$ is 0 and d is produced for this case also. So, a nonalternating output (d,d) is produced as a result of application of $(\omega,\overline{\omega})$ and $(\omega,\overline{\omega})$ therefore constitutes a test for f_g^d. Q.E.D.

As a result of the above theorem the alternating system shown in Figure 4.3 is self-checking for all single faults of the stuck-at type in both the alternating network, N, and its checker, C.

4.3. Two-Rail Alternating Checker

While the semi-adjoint exclusive-OR checker is an inherently simple checker, its use constrains the network structure in a manner that may be impractical in some cases. For this reason a checker which produces a fault indication when one or more checker inputs do not alternate is desirable. One such checker, Figure 4.4, utilizes an r-input totally self-checking (TSC) two-rail checker as a subnetwork. This two-rail checker has been studied in the literature [1] and has the following properties:

a) Inputs to the TSC two-rail checker are r input pairs, u_i, u_i', $1 \leq i \leq r$. Output from the checker is a single output pair v,v'.

b) The TSC two-rail checker is code disjoint. That is, if every input pair u_i, u_i' is two-rail $u_i' = \overline{u}_i$, $1 \leq i \leq r$, then the output v,v' is two-rail, $v' = \overline{v}$. But, if one or more input pairs are not two-rail then the output v,v' is not two-rail, $v' \neq \overline{v}$.

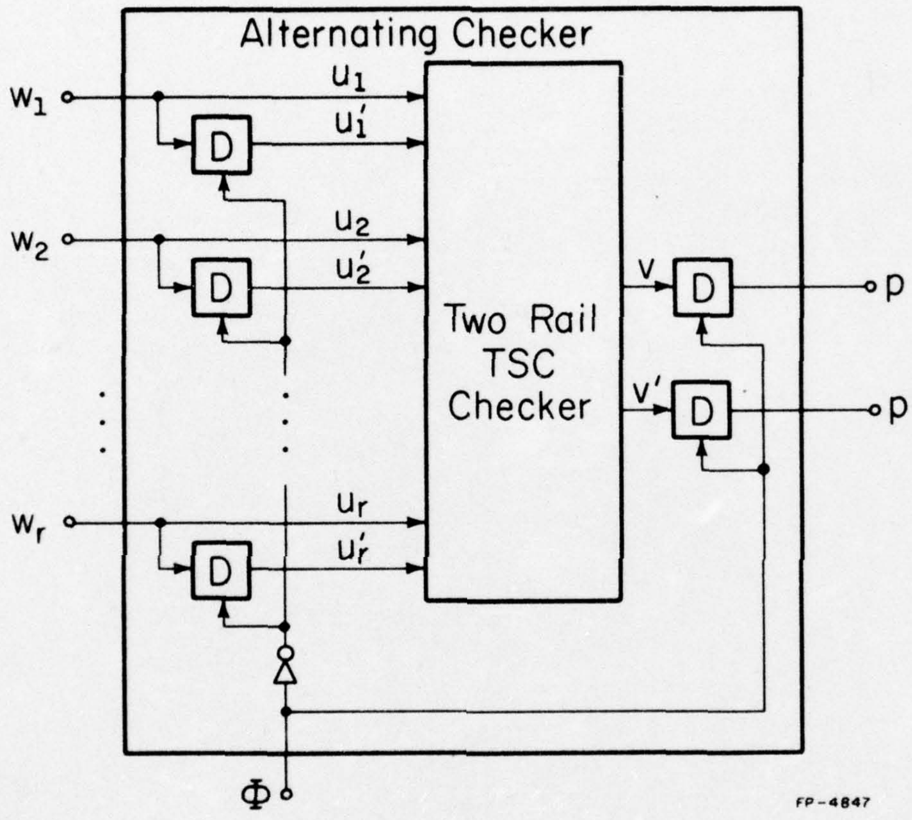


Figure 4.4. Two-rail alternating checker.

c) The TSC two-rail checker is self-testing. That is, for any stuck-at-d fault on any checker gate input or output there exists a set of r two-rail inputs which when applied to the two-rail checker produces a checker output which is not two-rail.

The two-rail checker when used as a subnetwork of the alternating checker is preceded by a set of D-flipflop memory elements. One such element is used on each input line so as to produce r input pairs from the r alternating inputs applied to the alternating checker. These r input pairs are then applied to the two-rail checker as u_i, u_i', $1 \leq i \leq r$. Thus, if r Φ-synchronized alternating inputs are applied, r two-rail pairs u_i, u_i', $u_i' = \overline{u}_i$, $1 \leq i \leq r$, are produced as input to the two-rail checker during the '0' of the clock. As a result, during that half-period, the two-rail checker output is also two-rail, $v' = \overline{v}$. If one or more of the r Φ-synchronized inputs are non-alternating, one or more of the two-rail checker inputs are not two-rail and the two-rail checker output v,v' during the '0' of the clock is not two-rail either, $v' \neq \overline{v}$. The two-rail checker outputs v,v' are latched using two additional D-flipflop memory elements at the beginning of each clock period. The resulting alternating checker output p,p' is a representation of v,v' with period T, where T is the period of the clock Φ. Specifically, during any clock period p,p' assumes the value on v,v' during the '0' of the clock in the previous period. Any fault detected by the alternating checker is thus indicated on p,p' as 0,0 or 1,1 in the clock period following the sensitization of the fault. A 0,1 or 1,0 on p,p' indicates normal operation during the previous clock period.

Theorem 4.4: The two-rail alternating checker is self-testing.

Proof: Consider any single stuck-at fault on lines in the two-rail TSC checker. For any such fault there exists a two-rail input $u = u_1u_2 \ldots u_r$ which tests for that fault as the two-rail TSC checker is itself self-testing. But then, input $(\omega,\overline{\omega}) = (u,\overline{u})$ must test for that fault in the alternating checker. That is, under alternating input $(\omega,\overline{\omega}) = (u,\overline{u})$, a fault indication, $v,v' = 0,0$ or $1,1$ is produced on the TSC two-rail checker output. That fault indication then appears on the output p,p' during the following clock period indicating the existence of the fault. Now consider any line fault stuck-at-d on memory elements preceding the two-rail TSC checker. Any such fault must manifest itself as $u_i,u_i' = d,d$ for any $(\omega,\overline{\omega})$ such that $(\omega_i,\overline{\omega}_i) = (\overline{d},d)$. However, if $u_i' \neq \overline{u}_i$ then $v' \neq \overline{v}$. Therefore $p' \neq \overline{p}$ in the following clock period and the fault is detected. Similarly, line faults on the output memory elements are also detected and the two-rail alternating checker is self-testing. Q.E.D.

Thus, any line fault on any memory element input or output or any line fault within the two-rail checker itself manifests itself as a fault signal. Any alternating system utilizing the two-rail alternating checker for monitoring network lines is therefore self-checking for all single faults.

The two-rail alternating checker is somewhat costly as a delay element must be added for each monitored network line. This restriction can be relaxed in the case of the synchronous machine. There, machine outputs $z_1,z_2,\ldots,z_\ell$ are monitored in normal fashion using a delay element in the two-rail alternating checker for each of the ℓ output lines. The feedback lines $Y_1, Y_2, \ldots, Y_m$ are applied to the checker as $u_{\ell+1}, u_{\ell+2}, \ldots, u_{\ell+m}$ and lines $y_1', y_2', \ldots, y_m'$ become $u_{\ell+1}', u_{\ell+2}', \ldots, u_{\ell+m}'$. (Note that $r = \ell+m$.) Thus, no delay elements are required in the two-rail alternating checker for $Y_1, Y_2, \ldots, Y_m$. A total of m delay elements is eliminated by utilizing delay elements already used in the synchronous machine.

5. COST OF ALTERNATING LOGIC IMPLEMENTATION

5.1. Introduction and Summary of Results

The implementation cost of any design method that yields networks with a measure of fault detection capability is of major concern. Results in this area are, however, somewhat sparse as design of fault tolerant digital systems is a relatively new field. Of those design methods where implementation cost is known, triple modular redundant (TMR) design and quadded logic design are most classic. In TMR design [11,16] basic circuitry is increased by a factor of 3. To this cost must be added the cost of the voter, which may appear as a single unit or may itself be triplicated. Implementation of quadded logic design [11,16] requires basic circuitry to be increased by a factor of 4 as each gate is quadrupled in the design. Both of the above methods are fault masking techniques.

Alternating logic design is a design method which yields totally self-checking digital networks which do not mask a fault but simply indicate its presence. In order to get a feel for its cost, self-checking alternating NOR networks representing all 218 three variable functions were obtained. These networks were compared with optimal NOR realizations [18] for each of the 218 three variable functions, obtaining in each case the ratio between the number of gates in the self-checking alternating realization to the number of gates in the optimal three variable realization. For the 218 functions examined these ratios varied from 1 to 7. Those functions with a ratio of 1 were the self-dual three variable functions whose optimal NOR realization had sufficient structural properties to be self-checking also. Eight of the 218 functions were in this class. Only one function had a ratio of 7. That function was $\overline{a+b+c}$, realizable as a simple NOR gate in a conventional design but requiring 7 gates in a self-checking alternating representation. Four of the 218 functions had a ratio greater than 3. The mean for all 218 functions was 1.85. This figure might reasonably be compared with the figure for duplicating machines, namely 2.

The mean self-checking alternating cost ratio, 1.85, obtained is not guaranteed to be optimal. That is, a self-checking alternating network may exist which represents a given three variable function while using less NOR gates than that network used in obtaining the above comparison. A lower bound to the mean, however, can be formulated from knowledge of the optimal NOR realization of the alternating network, not necessarily self-checking, representing each of the 218 three variable functions. To this end optimal NOR networks for all self-dual functions of 4 variables were obtained. These networks were associated with each of the 218 three variable functions they represent in order to form an optimal ratio in each case. The mean ratio, 1.63, was then calculated. Thus, the mean self-checking alternating cost ratio is bounded from below by 1.63.

The mean self-checking alternating cost ratio, 1.85, does not include the cost of the alternating logic checker. Any implementation of an alternating logic design must of course include this cost in its justification, just as the voter cost must be considered in TMR system design.

In the following sections the data supporting the results summarized here are presented, including NOR network diagrams for each optimal alternating network and each self-checking alternating network. If the details are not of interest, proceeding to Chapter 6 is advised.

5.2. Functional Notation

In order to define explicitly the manner in which the data are presented, it is probably best to pick a three variable function and find its optimal conventional NOR realization, its optimal alternating NOR realization, and the self-checking alternating NOR realization utilized for representing that function. To this end consider the function $f(x_1,x_2,x_3)$ specified in Figure 5.1a). The function can be represented as an 8-tuple $z_o,z_1,\ldots,z_7$ where $z_i = 0$ or 1, $0 \leq i \leq 7$, and where $z_o = f(0,0,0)$, $z_1 = f(0,0,1),\ldots,$ and $z_7 = f(1,1,1)$. That is, f can be represented as 00011100. As this notation is somewhat cumbersome a shorter notation based on the previous notation is used. The 8-tuple $z_oz_1\ldots z_7$ is parsed into two four-tuples, and a hexadecimal notation based on Table 5.1 is used. The resulting 2-tuple representation for the function is thus denoted $f(x_1,x_2,x_3) = y_oy_1$ where $y_i \in \{0,1,\ldots,9,A,B,\ldots,F\}$ for $0 \leq i \leq 1$. The function f is thus represented as $f(x_1,x_2,x_3) = 1C$, since the hexadecimal version of 0001 1100 is 1C.

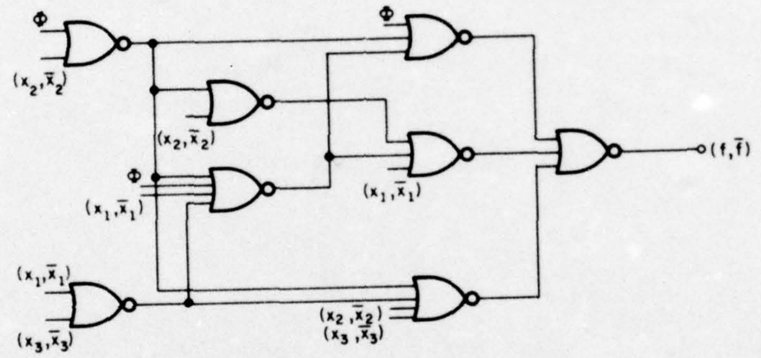
5.3. Results

Once the hexadecimal notation for a function f is obtained, Table 5.2 is consulted. The first column of that table lists the hexadecimal

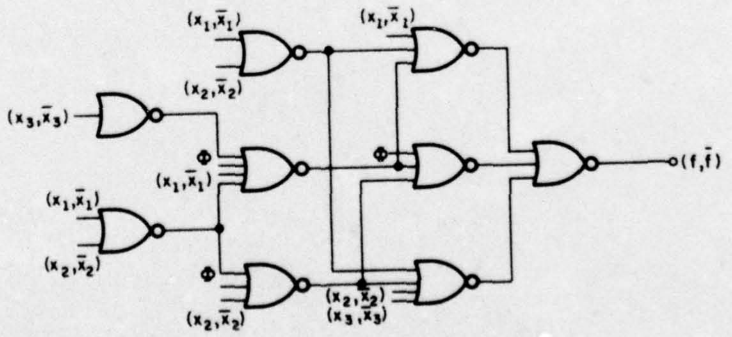
x_1 \ $x_2 x_3$	00	01	11	10
0	0	0	1	0
1	1	1	0	0

f

a)



b)



c)

FP-5085

Figure 5.1. Example 3 variable function.

Table 5.1. Hexadecimal Notation

z_i z_{i+1} z_{i+2} z_{i+3}	Y_j
0 0 0 0	0
0 0 0 1	1
0 0 1 0	2
0 0 1 1	3
0 1 0 0	4
0 1 0 1	5
0 1 1 0	6
0 1 1 1	7
1 0 0 0	8
1 0 0 1	9
1 0 1 0	A
1 0 1 1	B
1 1 0 0	C
1 1 0 1	D
1 1 1 0	E
1 1 1 1	F
$i = 0,4$	$j = i/4$

specifications for all 256 functions of three or fewer variables. The hexadecimal specification for the function in question is thus used to select a row of the table. The second column of the table lists the number of gates required in the conventional optimal NOR realization of each function as given in Hellerman [18]. (Complements of input variables are assumed to be unavailable.) Thus, for the function 1C, 6 NOR gates are required. A − in the second column indicates that the functional specification for that row is a specification for a 0, 1, or 2 variable function. The third through fifth columns contains data for the optimal alternating NOR network which represents f, namely the optimal network for the dualization of f, f_*, discussed in Chapter 2. For the function 1C there are 8 gates in this network, yielding a cost ratio of 8/6 or 1.33. The drawing numbers in the fourth column will be discussed later. The sixth through eighth columns contain data for an alternating network which represents f by realizing f_* but does so with sufficient structural properties to guarantee that it is a self-checking alternating network. For 1C this realization has 9 NOR gates, yielding a cost ratio of 9/6 or 1.5. A better realization for the self-checking alternating network may exist but at best it can have no fewer gates than the optimal alternating network, namely 8. The cost ratio 1.33 is thus a lower bound on the cost ratio for the function 1C.

The gates required in the optimal alternating networks representing f were obtained by finding optimal NOR realizations for the function f_* of each three variable function f. Optimal NOR realizations for every four variable self-dual function were thus required. Of the 3984 P (permutation of variable) equivalence classes of four variables, assuming complements of input variables are not available, 32 are P equivalence classes of self-dual functions. A representative of each of these 32 representative functions was selected and optimized by first minimizing the gate count, then minimizing the number of gate inputs, and, finally, minimizing the number of levels required to realize the function, utilizing a FORTRAN program 'ILLOD-(NOR-B)' written by Nakagawa and Lai [25]. (The optimizing algorithm used in that program is based on Davidson's [12] branch-and-bound method and is essentially a NOR version of Davidson's optimizing NAND decomposition algorithm.) After these optimal networks were produced, one was assigned to each three variable function along with a permutation number specifying how input variables are connected such that the network represents f by realizing f_*. Network drawings of the optimal alternating networks appear in Figure 5.2, p. 82. For the example function 1C, networks 22 and 23 are both optimal alternating networks for the function although neither is self-checking. Further, an input interconnection number, 1, is specified, delimited by a slash (/). Table 5.3, p. 80, associates this number with an input interconnection specification. Thus, for permutation 1, the clock Φ should be connected to a, x_1 to b, x_2 to d, and x_3 to c. The resulting network using optimal network number 22 is shown in Figure 5.1 b). It is the optimal alternating NOR network representing the function 1C.

A self-checking alternating network representing the same function can be obtained using the network drawing information in the seventh column of Table 5.2, p. 70. In this column is specified a network drawing number and an input interconnection specification as before. The network corresponding to these numbers appear in Figure 5.3, p. 84. For the example function, IC, network drawing number 22 is specified with interconnection number 1. The resulting self-checking alternating network is shown in Figure 5.1 c).

The self-checking alternating networks shown in Figure 5.3 were also generated by the FORTRAN program 'ILLOD-(NOR-B)'. The networks, however, were produced by the program in its search for the optimal networks. The self-checking capability of these networks was determined independently based on their structural properties.

Table 5.2. Cost Ratio and Network Drawing Table

	Realization						
	Optimal Conventional	Optimal Alternating			Self-checking Alternating		
Function	Gates	Gates	Drwg.	Ratio	Gates	Drwg.	Ratio
00	-	-	-	-	-	-	-
01	4	5	6/0	1.25	5	6/0	1.25
02	3	6	8/0	2.00	6	8/0	2.00
03	-	-	-	-	-	-	-
04	3	6	8/1	2.00	6	8/1	2.00
05	-	-	-	-	-	-	-
06	6	7	14/0	1.17	7	14/0	1.17
07	3	5	6/6	1.67	5	6/6	1.67
08	2	6	9/0	3.00	7	9/0	3.50
09	5	7	14/6	1.40	7	14/6	1.40
0A	-	-	-	-	-	-	-
0B	4	6	8/6	1.50	6	8/6	1.50
0C	-	-	-	-		-	-
0D	4	6	8/7	1.50	6	8/7	1.50
0E	5	6	9/6	1.20	7	9/6	1.40
0F	-	-	-	-	-	-	-
10	3	6	8/4	2.00	6	8/4	2.00
11	-	-	-	-	-	-	-
12	6	7	14/2	1.17	7	14/2	1.17
13	3	5	6/8	1.67	5	6/8	1.67
14	6	7	14/3	1.17	7	14/3	1.17
15	3	5	6/9	1.67	5	6/9	1.67
16	7	8	20/0	1.14	9	20/0	1.28
17	4	4	3/18	1.00	4	3/18	1.00
18	6	8	21/0	1.33	9	21/0	1.50
19	5	8	20/8	1.80	9	20/8	1.80
1A	6	8	22,23/0	1.33	9	22/0	1.50

Table 5.2 continued

	Realization						
	Optimal Conventional	Optimal Alternating			Self-Checking Alternating		
Function	Gates	Gates	Drwg.	Ratio	Gates	Drwg.	Ratio
1B	4	7	14/12	1.75	7	14/12	1.75
1C	6	8	22,23/1	1.33	9	22/1	1.50
1D	4	7	14/13	1.75	7	14/13	1.75
1E	6	8	21/6	1.33	9	21/6	1.50
1F	3	6	8/18	2.00	6	8/18	2.00
20	2	6	9/2	3.00	7	9/2	3.50
21	5	7	14/8	1.40	7	14/8	1.40
22	-	-	-	-	-	-	-
23	4	6	8/8	1.50	6	8/8	1.50
24	6	8	21/2	1.33	9	21/2	1.50
25	5	8	20/8	1.60	9	20/8	1.80
26	6	8	22,23/2	1.33	9	22/2	1.50
27	4	7	14/14	1.75	7	14/14	1.75
28	5	7	15/0	1.40	8	15/0	1.60
29	6	8	22,23/12	1.33	9	22/12	1.50
2A	2	6	10/0	3.00	6	10/0	3.00
2B	5	5	5/21	1.00	5	5/21	1.00
2C	5	8	24/0	1.60	9	24/0	1.80
2D	5	8	21/12	1.60	9	21/2	1.80
2E	5	7	15/6	1.40	8	15/6	1.60
2F	4	6	9/12	1.50	7	9/12	1.75
30	-	-	-	-	-	-	-
31	4	6	8/10	1.50	6	8/10	1.50
32	5	6	9/8	1.20	7	9/8	1.40
33	-	-	-	-	-	-	-
34	6	8	22,23/4	1.33	9	22/4	1.50
35	4	7	14/16	1.75	7	14/16	1.75

Table 5.2 Continued

	Realization						
	Optimal Conventional	Optimal Alternating			Self-Checking Alternating		
Function	Gates	Gates	Drwg.	Ratio	Gates	Drwg.	Ratio
36	6	8	21/8	1.33	9	21/8	1.50
37	3	6	8/20	2.00	6	8/20	2.00
38	5	8	24/2	1.60	9	23/2	1.80
39	5	8	21/14	1.60	9	21/14	1.80
3A	5	7	15/8	1.40	8	15/8	1.60
3B	4	6	9/14	1.50	7	9/14	1.75
3C	-	-	-	-	-	-	-
3D	5	8	22,23/18	1.60	9	22/18	1.80
3E	6	8	24/12	1.33	9	23/12	1.50
3F	-	-	-	-	-	-	-
40	2	6	9/3	3.00	7	9/3	3.50
41	5	7	12/9	1.40	7	12/9	1.40
42	6	8	21/3	1.33	9	21/3	1.50
43	5	8	20/9	1.60	9	20/9	1.80
44	-	-	-	-	-	-	-
45	4	6	8/9	1.50	6	8/9	1.50
46	6	8	22,23/3	1.33	9	22/3	1.50
47	4	7	14/15	1.75	7	14/15	1.75
48	5	7	15/1	1.40	8	15/1	1.60
49	6	8	22,23/13	1.33	9	22/13	1.50
4A	5	8	24/1	1.60	9	23/1	1.80
4B	5	8	21/13	1.60	9	21/13	1.80
4C	2	6	10/1	3.00	6	10/1	3.00
4D	5	5	5/20	1.00	5	5/20	1.00
4E	5	7	15/7	1.40	8	15/7	1.60
4F	4	6	9/13	1.50	7	9/13	1.75

Table 5.2 Continued

	Realization						
	Optimal Conventional	Optimal Alternating			Self-Checking Alternating		
Function	Gates	Gates	Drwg.	Ratio	Gates	Drwg.	Ratio
50	-	-	-	--	-	-	-
51	4	6	8/11	1.50	6	8/11	1.50
52	6	8	22,23/5	1.33	9	22/5	1.50
53	4	7	14/17	1.75	7	14/17	1.75
54	5	6	9/9	1.20	7	9/9	1.40
55	-	-	-	-	-	-	-
56	6	8	21/9	1.33	9	21/9	1.50
57	3	6	8/21	2.00	6	8/21	2.00
58	5	8	24/3	1.60	9	23/3	1.80
59	5	8	21/15	1.60	9	21/15	1.80
5A	-	-	-	-	-	-	-
5B	5	8	22,23/19	1.60	9	22/19	1.80
5C	5	7	15/9	1.40	8	15/9	1.60
5D	4	6	9/15	1.50	7	9/15	1.75
5E	6	8	24/13	1.33	9	23/13	1.50
5F	-	-	-	-	-	-	-
60	5	7	15/4	1.40	8	15/4	1.60
61	6	8	22,23/16	1.33	9	22/16	1.50
62	5	8	24/4	1.60	9	23/4	1.80
63	5	8	21/16	1.60	9	21/16	1.80
64	5	8	24/5	1.60	9	23/5	1.80
65	5	8	21/17	1.60	9	21/17	1.80
66	-	-	-	-	-	-	-
67	5	8	22,23/22	1.60	9	22/22	1.80
68	6	8	29,30/0	1.33	9	28/0	1.50
69	7	7	13/18	1.00	8	13/18	1.14

Table 5.2 Continued

	Realization						
	Optimal Conventional	Optimal Alternating			Self-Checking Alternating		
Function	Gates	Gates	Drwg.	Ratio	Gates	Drwg.	Ratio
6A	5	8	31/0	1.60	10	29/0	2.00
6B	6	8	24/18	1.33	9	23/18	1.50
6C	5	8	31/1	1.60	10	29/1	2.00
6D	6	8	24/19	1.33	9	23/19	1.50
6E	5	8	29,30/6	1.60	9	28/6	1.80
6F	5	7	15/18	1.40	8	15/18	1.60
70	2	6	10/4	3.00	6	10/4	3.00
71	5	5	5/18	1.00	5	5/18	1.00
72	5	7	15/10	1.40	8	15/10	1.60
73	4	6	9/18	1.50	7	9/18	1.75
74	5	7	15/11	1.40	8	15/11	1.60
75	4	6	9/17	1.50	7	9/17	1.75
76	6	8	24/16	1.33	9	23/16	1.50
77	-	-	-	-	-	-	-
78	5	8	31/4	1.60	10	29/4	2.00
79	6	8	24/22	1.33	9	23/22	1.50
7A	5	8	29,30/8	1.60	9	28/8	1.80
7B	5	7	15/20	1.40	8	15/20	1.60
7C	5	8	29,30/9	1.60	9	28/9	1.80
7D	5	7	15/21	1.40	8	15/21	1.60
7E	6	8	31/18	1.33	10	29/18	1.67
7F	2	6	10/18	3.00	6	10/18	3.00
80	1	7	16/0	7.00	7	16/0	7.00
81	5	8	25/0	1.60	11	24/0	2.20
82	4	7	17/0	1.75	9	17/0	2.25
83	4	8	32/0	2.00	10	30/0	2.50
84	4	7	17/1	1.75	9	17/1	2.25

Table 5.2 Continued

	Realization						
	Optimal Conventional	Optimal Alternating			Self-Checking Alternating		
Function	Gates	Gates	Drwg.	Ratio	Gates	Drwg.	Ratio
85	4	8	32/1	2.00	10	30/1	2.50
86	7	8	33/0	1.14	10	31/0	1.42
87	4	8	25/6	2.00	11	24/6	2.75
88	-	-	-	-	-	-	-
89	5	8	33/6	1.60	10	31/6	2.00
8A	3	7	18/0	2.33	8	18/0	2.67
8B	4	7	17/6	1.75	9	17/6	2.25
8C	3	7	18/1	2.33	8	18/1	2.67
8D	4	7	17/7	1.75	9	17/7	2.25
8E	6	6	7/22	1.00	6	7/22	1.00
8F	3	7	16/6	2.33	7	16/6	2.33
90	4	7	17/4	1.75	9	17/4	2.25
91	4	8	32/4	2.00	10	30/4	2.50
92	7	8	33/2	1.14	10	31/2	1.42
93	4	8	25/8	2.00	11	24/8	2.75
94	7	8	33/3	1.14	10	31/3	1.42
95	4	8	25/9	2.00	11	24/9	2.75
96	7	7	12/18	1.00	9	12/18	1.28
97	5	8	32/18	1.60	10	30/18	2.00
98	6	8	34,35/0	1.33	10	32/0	1.67
99	-	-	-	-	-	-	-
9A	6	8	26/0	1.33	10	25/0	1.67
9B	4	8	33/12	2.00	10	31/12	2.50
9C	6	8	26/1	1.33	10	25/1	1.67
9D	4	8	33/13	2.00	10	31/13	2.50
9E	7	8	34,35/6	1.14	10	32/6	1.42
9F	4	7	17/18	1.75	9	17/18	2.25

Table 5.2 Continued

	Realization						
	Optimal Conventional	Optimal Alternating			Self-Checking Alternating		
Function	Gates	Gates	Drwg.	Ratio	Gates	Drwg.	Ratio
A0	-	-	-	-	-	-	-
A1	5	8	33/8	1.60	10	31/8	2.00
A2	3	7	18/2	2.33	8	18/2	2.67
A3	4	7	17/8	1.75	9	17/8	2.25
A4	6	8	34,35/2	1.33	10	32/2	1.67
A5	-	-	-	-	-	-	-
A6	6	8	26/2	1.33	10	25/2	1.67
A7	4	8	33/14	2.00	10	31/14	2.50
A8	4	7	19/0	1.75	7	19/0	1.75
A9	6	8	26/12	1.33	10	25/12	1.67
AA	-	-	-	-	-	-	-
AB	4	7	18/12	1.75	8	18/12	2.00
AC	5	8	27/0	1.60	9	26/0	1.80
AD	5	8	34,35/12	1.60	10	32/12	2.00
AE	5	7	19/6	1.40	7	19/6	1.40
AF	-	-	-	-	-	-	-
B0	3	7	18/4	2.33	8	18/4	2.67
B1	4	7	17/10	1.75	9	17/10	2.25
B2	6	6	7/19	1.00	6	7/19	1.00
B3	3	7	16/8	2.33	7	16/8	2.33
B4	6	8	26/4	1.33	10	25/4	1.67
B5	4	8	33/16	2.00	10	31/16	2.50
B6	7	8	34,35/8	1.14	10	32/8	1.42
B7	4	7	17/20	1.75	9	17/20	2.25
B8	5	8	27/2	1.60	9	26/2	1.80
B9	5	8	34,35/14	1.60	10	32/14	2.00

Table 5.2 Continued

	Realization						
	Optimal Conventional	Optimal Alternating			Self-Checking Alternating		
Function	Gates	Gates	Drwg.	Ratio	Gates	Drwg.	Ratio
BA	5	7	19/8	1.40	7	19/8	1.40
BB	-	-	-	-	-	-	-
BC	6	8	28/0	1.33	11	27/0	1.83
BD	5	8	26/18	1.60	10	25/18	2.00
BE	6	8	27/12	1.33	9	26/12	1.50
BF	3	7	18/18	2.33	8	18/18	2.67
C0	-	-	-	-	-	-	-
C1	5	8	33/9	1.60	10	31/9	2.00
C2	3	8	34,35/3	2.67	10	32/3	3.33
C3	-	-	-	-	-	-	-
C4	3	7	18/3	2.33	8	18/3	2.67
C5	4	7	17/9	1.75	9	17/9	2.25
C6	6	8	26/3	1.33	10	25/3	1.67
C7	4	8	33/15	2.00	10	31/15	2.50
C8	4	7	19/1	1.75	7	19/1	1.75
C9	6	8	26/13	1.33	10	25/13	1.67
CA	5	8	27/1	1.60	9	26/1	1.80
CB	5	8	34,35/13	1.60	10	32/13	2.00
CC	-	-	-	-	-	-	-
CD	4	7	18/13	1.75	8	18/13	2.00
CE	5	7	19/27	1.40	7	19/27	1.40
CF	-	-	-	-	-	-	-
D0	3	7	18/5	2.33	8	18/5	2.67
D1	4	7	17/11	1.75	9	17/11	2.25
D2	6	8	26/5	1.33	10	25/5	1.67
D3	4	8	33/17	2.00	10	31/17	2.50
D4	6	6	7/18	1.00	6	7/18	1.00

Table 5.2 Continued

	Realization						
	Optimal Conventional	Optimal Alternating			Self-Checking Alternating		
Function	Gates	Gates	Drwg.	Ratio	Gates	Drwg.	Ratio
D5	5	7	16/9	1.40	7	16/9	1.40
D6	7	8	34,35/9	1.14	10	32/9	1.42
D7	4	7	17/21	1.75	9	17/21	2.25
D8	5	8	27/13	1.60	9	26/13	1.80
D9	5	8	34,35/15	1.60	10	32/15	2.00
DA	6	8	28/1	1.33	11	27/1	1.83
DB	5	8	26/19	1.60	10	25/19	2.00
DC	5	7	19/9	1.40	7	19/9	1.40
DD	-	-	-	-	-	-	-
DE	6	8	27/13	1.33	9	26/13	1.80
DF	3	7	18/19	2.33	8	18/19	2.67
E0	4	7	19/4	1.75	7	19/4	1.75
E1	6	8	26/16	1.33	10	25/16	1.67
E2	5	8	27/4	1.60	9	26/4	1.80
E3	5	8	34,35/16	1.60	10	32/16	2.00
E4	5	8	27/5	1.60	9	26/5	1.80
E5	5	8	34,35/17	1.60	10	32/17	2.00
E6	6	8	28/4	1.33	11	27/4	1.83
E7	5	8	26/21	1.60	10	25/21	2.00
E8	5	5	4/18	1.00	5	4/18	1.00
E9	7	8	28/18	1.14	11	27/18	1.57
EA	4	6	11/0	1.50	6	11/0	1.50
EB	5	8	27/18	1.60	9	26/18	1.80
EC	4	6	11/1	1.50	6	11/1	1.50
ED	5	8	27/19	1.60	9	26/19	1.80
EE	-	-	-	-	-	-	-
EF	4	7	19/18	1.75	7	19/18	1.75

Table 5.2 Concluded

	Realization						
	Optimal Conventional	Optimal Alternating			Self-Checking Alternating		
Function	Gates	Gates	Drwg.	Ratio	Gates	Drwg.	Ratio
F0	-	-	-	-	-	-	-
F1	4	7	18/16	1.75	8	18/16	2.00
F2	5	7	19/10	1.40	7	19/10	1.40
F3	-	-	-	-	-	-	-
F4	5	7	19/11	1.40	7	19/11	1.40
F5	-	-	-	-	-	-	-
F6	6	8	27/16	1.33	9	26/16	1.80
F7	3	7	18/22	2.33	8	18/22	2.67
F8	4	6	11/4	1.50	6	11/4	1.50
F9	5	8	27/22	1.60	9	26/22	1.80
FA	-	-	-	-	-	-	-
FB	4	7	19/20	1.75	7	19/20	1.75
FC	-	-	-	-	-	-	-
FD	4	7	19/21	1.75	7	19/21	1.75
FE	5	6	11/18	1.20	6	11/18	1.20
FF	-	-	-	-	-	-	-

Table 5.3. Input Interconnection Specification

Permutation Number	Variables			
	Φ	x_1	x_2	x_3
0	a	b	c	d
1	a	b	d	c
2	a	c	b	d
3	a	c	d	b
4	a	d	b	c
5	a	d	c	b
6	b	a	c	d
7	b	a	d	c
8	c	a	b	d
9	b	c	d	a
10	b	d	a	c
11	b	d	c	a
12	b	c	a	d
13	c	a	d	b
14	c	b	a	d
15	c	b	d	a
16	c	d	a	b
17	c	d	b	a
18	d	a	b	c
19	d	a	c	b
20	d	b	a	c
21	d	b	c	a
22	d	c	a	b
23	d	c	b	a

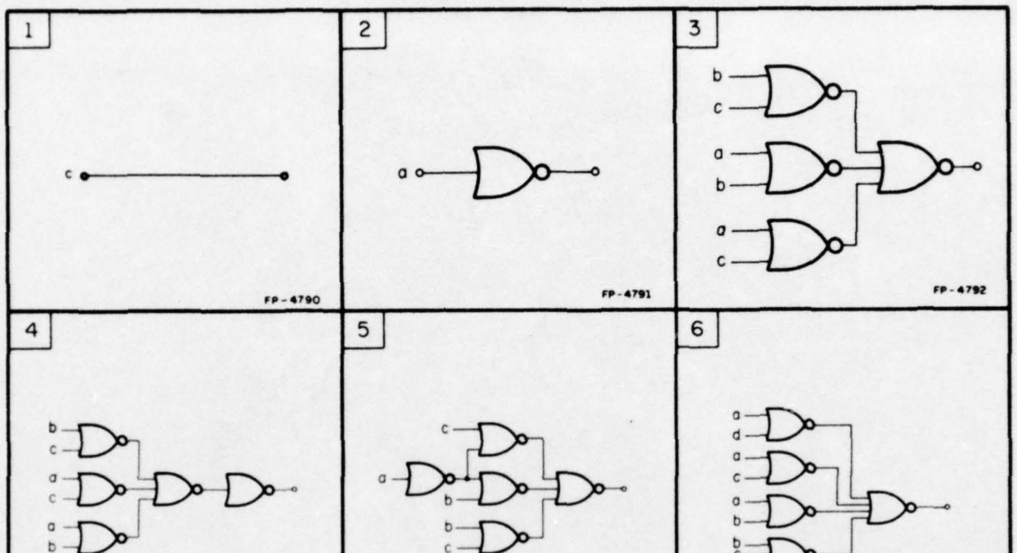


Figure 5.2. Optimal alternating NOR networks.

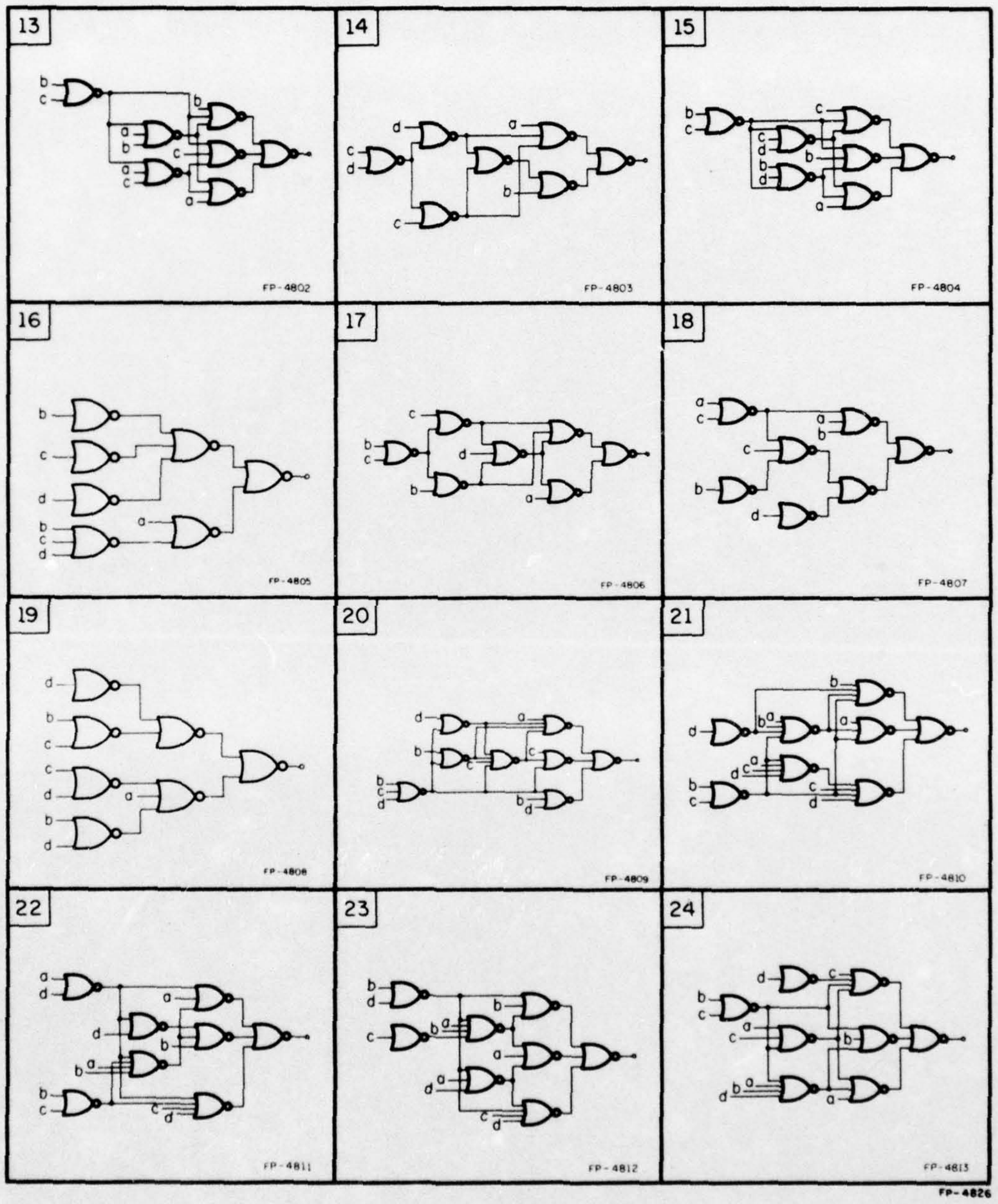


Figure 5.2. continued.

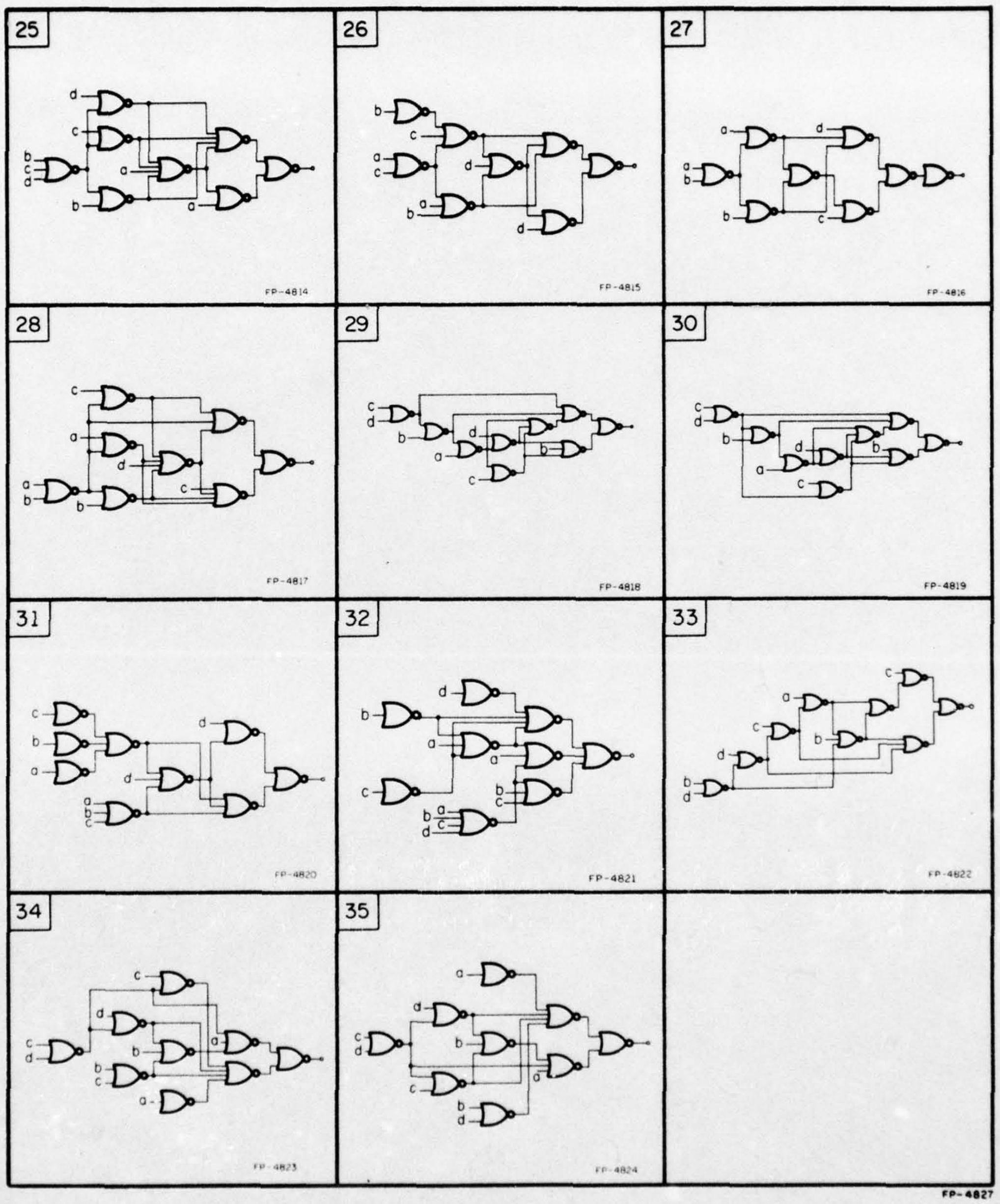


Figure 5.2. concluded.

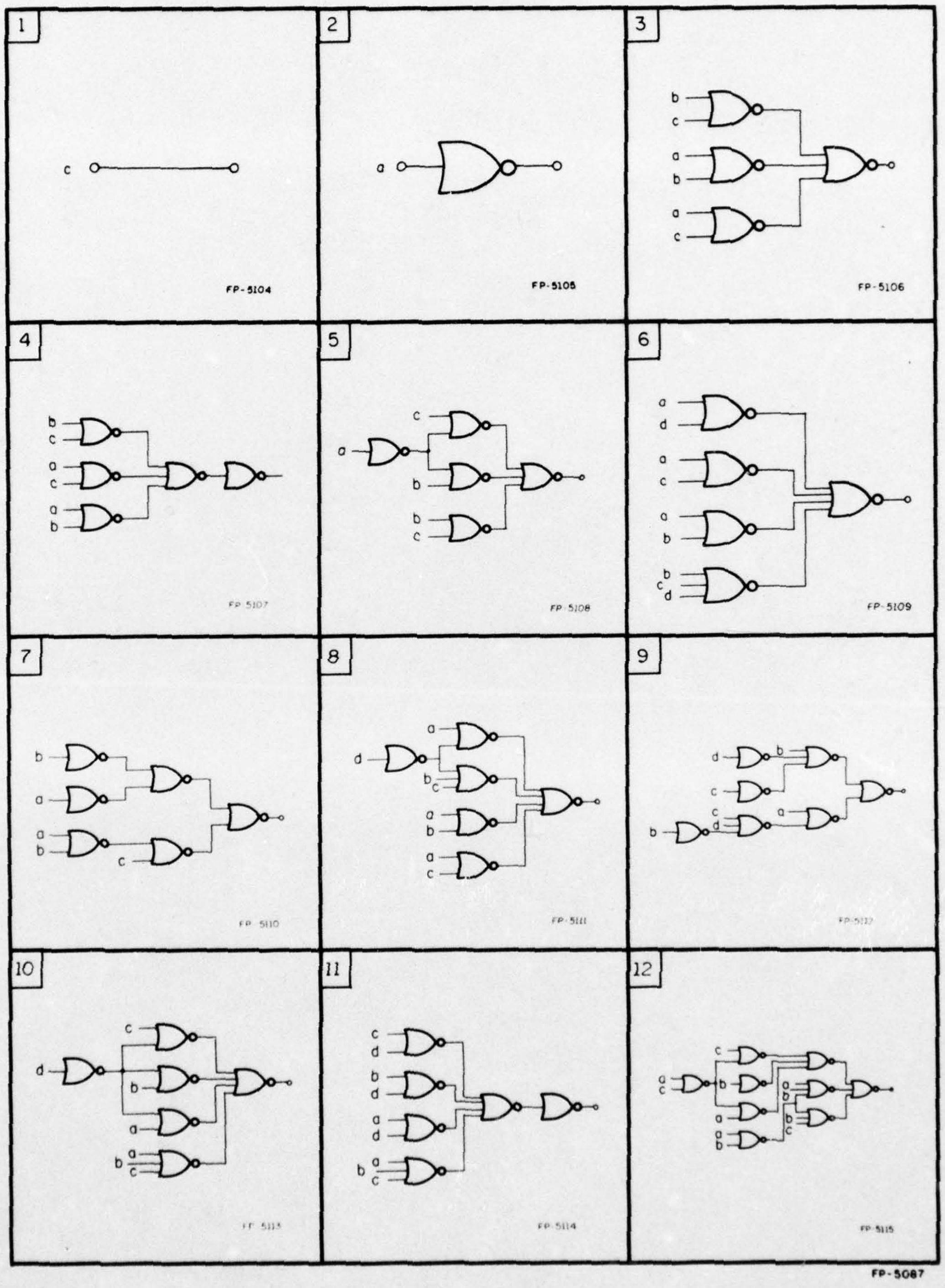


Figure 5.3. Self-checking alternating NOR networks.

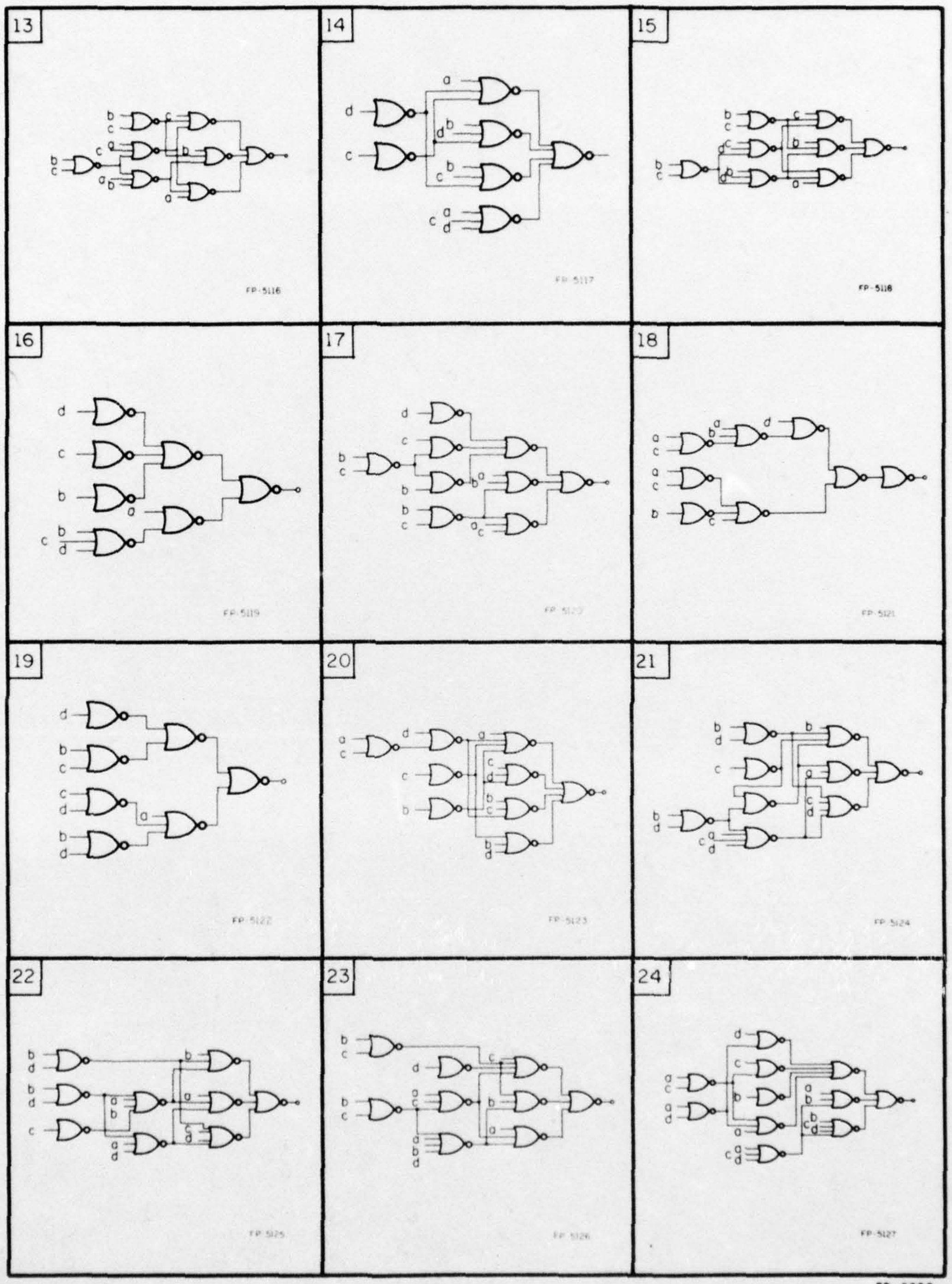


Figure 5.3. continued.

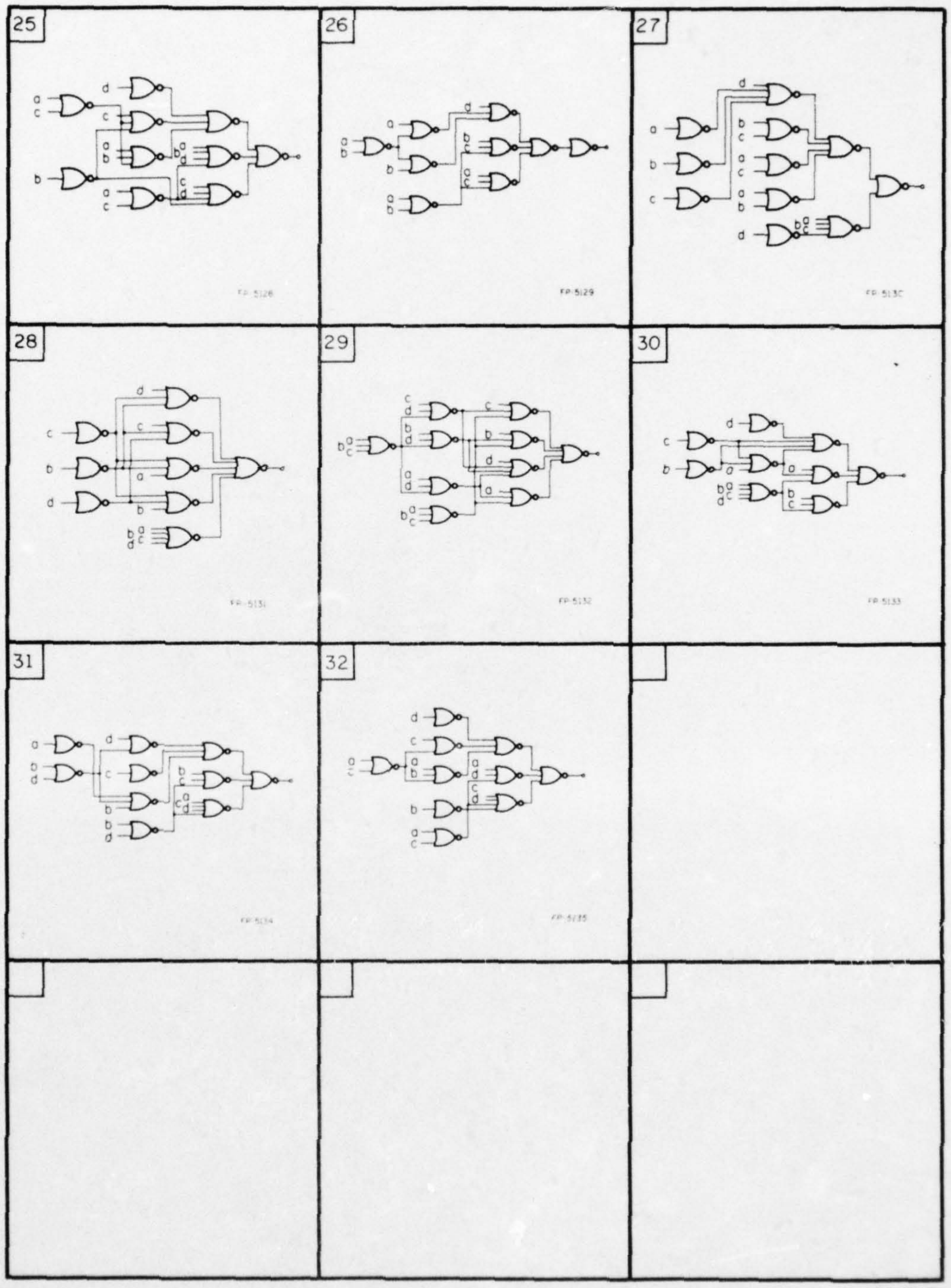
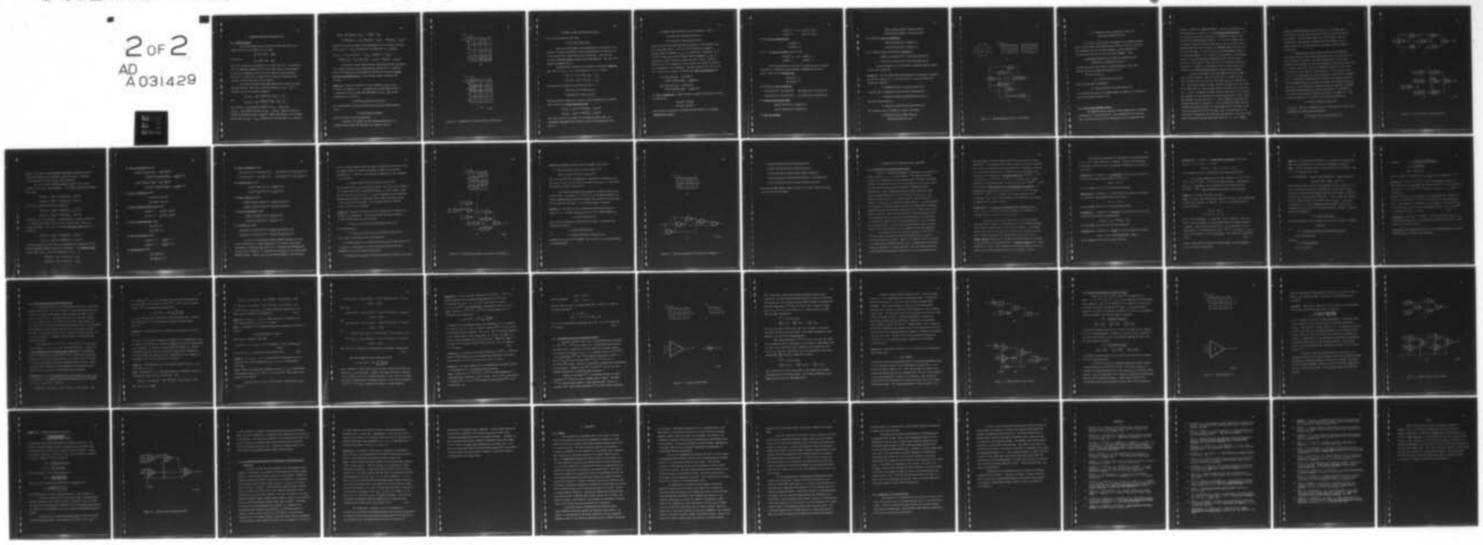


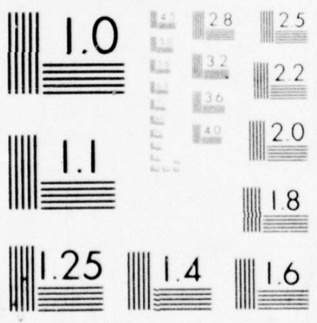
Figure 5.3. Concluded.

AD-A031 429 ILLINOIS UNIV AT URBANA-CHAMPAIGN COORDINATED SCIENCE LAB F/G 9/2
THE DESIGN OF ALTERNATING LOGIC SYSTEMS WITH FAULT DETECTION CA--ETC(U)
AUG 76 D A REYNOLDS DAAB07-72-C-0259
UNCLASSIFIED R-738 NL

2 OF 2
AD
A031429



END
DATE
FILMED
11-76



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963

6. ALTERNATING DESIGN WITH NAND/NOR LOGIC

6.1. NAND/NOR Algebra

The two-input NAND function and the two-input NOR function are commonly denoted by such symbols as the stroke

$$a|b = \overline{ab} = \bar{a} + \bar{b} \quad \text{(NAND)}$$

and the arrow

$$a{\downarrow}b = \overline{a+b} = \bar{a}\,\bar{b} \quad \text{(NOR)}.$$

While such a notation is convenient for two input functions, its implementation in a NAND/NOR decomposition of a multi-input function becomes awkward to read and to manipulate, and an alternate notation becomes desirable. Hohn [19] has proposed such a notation and developed an algebra for manipulating a function expressed in that notation.

Let M denote the multi-input NAND function and m denote the multi-input NOR function. Then for all switching functions $f_1, f_2, \ldots, f_k$ of n variables, and for all possible integers k,

$$M(f_1,f_2,\ldots,f_k) = \overline{f_1 f_2 \ldots f_k} = \bar{f}_1+\bar{f}_2 + \cdots + \bar{f}_k$$

and

$$m(f_1,f_2,\ldots,f_k) = \overline{f_1+f_2 + \cdots + f_k} = \bar{f}_1\bar{f}_2\ldots\bar{f}_k.$$

Thus, $M(f_1,f_2,\ldots,f_k)$ denotes the NAND of $f_1,f_2,\ldots,$ and f_k and $m(f_1,f_2,\ldots,f_k)$ denotes the NOR of $f_1,f_2,\ldots,$ and f_k. When this notation is applied to the familiar sum-of-products to NAND transformation, the following results where $g_{i1}\ g_{i2}\ \ldots\ g_{ir_i}$ represents the i-th implicant of r_i variables.

$$g_{11}g_{12}\cdots g_{1r_1}+g_{21}g_{22}\cdots g_{2r_2}+\cdots+g_{k1}g_{k2}\cdots g_{kr_k}$$

$$= M[M(g_{11},g_{12},\ldots,g_{1r_1}),M(g_{21},g_{22},\ldots,g_{2r_2}),\ldots,M(g_{k1},g_{k2},\ldots,g_{kr_k})].$$

Similarly the product-of-sums to NOR transformation can be written as follows where $h_{i1}+h_{i2}+\cdots+h_{ir_i}$ represents the i-th implicate of r_i variables.

$$(h_{11}+h_{12}+\cdots+h_{1r_1})(h_{21}+h_{22}+\cdots+h_{2r_2})\cdots(h_{k1}+h_{k2}+\cdots+h_{kr_k})$$

$$= m[m(h_{11},h_{12},\ldots,h_{1r_1}),m(h_{21},h_{22},\ldots,h_{2r_2}),\ldots,m(h_{k1},h_{k2},\ldots,h_{kr_k})].$$

Utilization of these transformations enable NAND and NOR expressions to be obtained directly from the Karnaugh map specification of a function. Further, the NAND and NOR representations obtained are necessarily <u>minimal two-level representations of f</u>. The following example illustrates the procedure.

<u>Example 6.1</u>: Consider a function f of four variables x_1,x_2,x_3, and x_4 specified as in Figure 6.1 a). A minimal two-level NOR decomposition of f is obtained by first circling prime implicates as shown. Then the NOR decomposition is written as,

$$f = m[m(\bar{x}_1,x_2),m(\bar{x}_1,\bar{x}_3),m(x_2,\bar{x}_3,\bar{x}_4)].$$

This representation is really obtained from the product-of-sums representation,

$$f = (\bar{x}_1+x_2)(\bar{x}_1+\bar{x}_3)(x_2+\bar{x}_3+\bar{x}_4)$$

but can be written from the map directly.

Similarly, the minimal two-level NAND decomposition of f is obtained from the prime implicant map of f, Figure 6.1 b), as

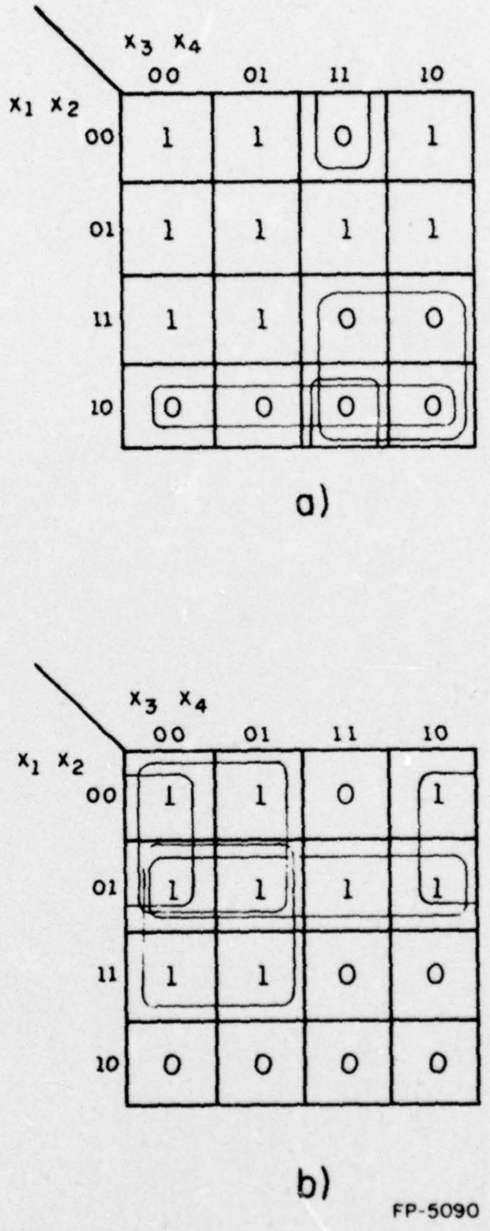


Figure 6.1. Karnaugh maps for NAND and NOR decompositions.

$$f = M[M(\overline{x}_1,x_2),M(\overline{x}_1,\overline{x}_3),M(\overline{x}_1,\overline{x}_4),M(x_2,\overline{x}_3)],$$

with the sum-of-products form being

$$f = \overline{x}_1x_2 + \overline{x}_1\overline{x}_3 + \overline{x}_1\overline{x}_4 + x_2\overline{x}_3.$$

Once the two-level NAND or NOR decompositions of a function f are obtained, it should be possible to effect transformations on these so as to produce possibly more optimal multi-level realizations. Hohn [19] has developed a NAND/NOR algebra to handle such transformations. The laws of the algebra are summarized here.

The NAND and NOR operations are operations obeying the commutative law. Thus, if $i_1,i_2,\ldots,i_k$ is any permutation of $1,2,\ldots,k$ then

$$M(f_1,f_2,\ldots,f_k) = M(f_{i_1},f_{i_2},\ldots,f_{i_k})$$
$$m(f_1,f_2,\ldots,f_k) = m(f_{i_1},f_{i_2},\ldots,f_{i_k}).$$

The operations are however not associative; that is,

$$M[f_1,M(f_2,f_3)] \neq M[M(f_1,f_2),f_3]$$
$$m[f_1,m(f_2,f_3)] \neq m[m(f_1,f_2),f_3].$$

When more than one operation is involved, certain regroupings are possible. These result in the hybrid associative laws.

$$M(f_1,f_2,\ldots,f_k,h^*) = M[m(\overline{f}_1,\overline{f}_2,\ldots,\overline{f}_k),h^*]$$
$$m(f_1,f_2,\ldots,f_k,h^*) = m[M(\overline{f}_1,\overline{f}_2,\ldots,\overline{f}_k),h^*]$$

These laws turn out to be useful for utilizing available inputs, for eliminating complements when desirable, and for controlling fan-in and fan-out.

The NAND and NOR operations are not distributive. That is,

$$M(f_1,m(f_2,f_3)) \neq m(M(f_1,f_2),M(f_1,f_3))$$

$$m(f_1,M(f_2,f_3)) \neq M(m(f_1,f_2),m(f_1,f_3)).$$

However, the factoring laws play the role played by the distributive laws in the factoring process. Let f^*, g^*, and h^* denote any nonnegative integral number of arguments f_i, g_i, h_i, respectively. Similarly, let f^+, g^+, and h^+ denote any positive integral number of arguments f_i, g_i, h_i, respectively. Thus, h^* may be an empty set, but h^+ cannot. Moreover, whenever h^* or h^+ is repeated in a given identity, it stands for the same set of arguments at each appearance. The same applies for f^*, f^+, g^*, and g^+. Utilizing this notation, the generalized <u>laws of factorization</u> are

$$M[M(f^+,g_1^+),M(f^+,g_2^+),\ldots,M(f^+,g_k^+),h^*]$$
$$= M[M(f^+,M\{M(g_1^+),M(g_2^+),\ldots,M(g_k^+)\}),h^*]$$

$$m[m(f^+,g_1^+),m(f^+,g_2^+),\ldots,m(f^+,g_k^+),h^*]$$
$$= m[m(f^+,m\{m(g_1^+),m(g_2^+),\ldots,m(g_k^+)\}),h^*].$$

The <u>*laws of idempotency*</u> *for NAND and NOR are* essentially derived from those for AND and OR.

$$M(f,f,g^*) = M(f,g^*)$$

$$m(f,f,g^*) = m(f,g^*)$$

From the basic definitions of the NAND and NOR operations, the <u>laws of operation with 0 and 1</u>,

$$M(0,g^*) = 1 \quad \text{and} \quad M(1,f^+) = M(f^+)$$
$$m(1,g^*) = 0 \qquad m(0,f^+) = m(f^+),$$

and the <u>laws of complementarity</u>,

$$M(f,\overline{f},g^*) = 1$$
$$m(f,\overline{f},g^*) = 0$$

result. The <u>laws of involution</u> take the form

$$M(M(f)) = f \quad \text{and} \quad m(M(f)) = f$$
$$m(m(f)) = f \qquad M(m(f)) = f.$$

Other transformations or laws which are for the most part simply translations of AND/OR/NOT transformations to NAND/NOR form also are useful. These are the <u>reduction laws</u>

$$M(f,m(f,g)) = 1$$
$$m(f,M(f,g)) = 0,$$

the generalized <u>laws of redundancy</u>,

$$M[f,M(f,g^+),h^*] = M[f,M(g^+),h^*] \qquad M[f,m(\overline{f},g^+),h^*] = M[f,m(g^+),h^*]$$
$$m[f,m(f,g^+),h^*] = m[f,m(g^+),h^*] \qquad m[f,M(\overline{f},g^+),h^*] = m[f,M(g^+),h^*]$$

the <u>generalized absorption laws</u>,

$$M[M(f^+),M(f^+,g^*),h^*] = M[M(f^+),h^*]$$
$$m[m(f^+),m(f^+,g^*),h^*] = m[m(f^+),h^*],$$

the <u>laws of consensus</u>,

$$M[M(f,g),M(\bar{f},h),M(g,h)] = M[M(f,g),M(\bar{f},h)]$$

$$m[m(f,g),m(\bar{f},h),m(g,h)] = m[m(f,g),m(\bar{f},h)],$$

the generalized <u>laws of condensation</u>,

$$M[M(f,g^{+}),M(\bar{f},g^{+}),h^{*}] = M[M(g^{+}),h^{*}]$$

$$m[m(f,g^{+}),m(\bar{f},g^{+}),h^{*}] = m[m(g^{+}),h^{*}],$$

and a somewhat useful transformation, <u>identity I1</u>,

$$M[M\{M(f_1,f_2),M(\bar{f}_1,\bar{f}_2)\},h^{*}] = M[M(f_1,\bar{f}_2),M(\bar{f}_1,f_2),h^{*}]$$

$$m[m\{m(f_1,f_2),m(\bar{f}_1,\bar{f}_2)\},h^{*}] = m[m(f_1,\bar{f}_2),m(\bar{f}_1,f_2),h^{*}].$$

As an illustration of the application of NAND/NOR algebra consider the full adder, Hohn [19].

<u>Example 6.2</u>: The full adder model and Karnaugh maps specifying its operation are shown in Figure 6.2 a). From the map for S, a two-level NAND representation is obtained,

$$S = M[M(\bar{a},\bar{b},c_i),M(\bar{a},b,\bar{c}_i),M(a,\bar{b},\bar{c}_i),M(a,b,c_i)].$$

Using the laws of commutativity and the laws of factorization,

$$S = M[M(a,\bar{b},\bar{c}_i),M(\bar{a},b,\bar{c}_i),M(M\{M(\bar{a},\bar{b}),M(a,b)\},c_i)].$$

But then, using identity I1,

$$S = M[M(a,\bar{b},\bar{c}_i),M(\bar{a},b,\bar{c}_i),M\{M(a,\bar{b}),M(\bar{a},b),c_i\}].$$

Now, using the laws of redundancy to remove complements

$$S = M[M\{a,M(b),M(c_i)\},M\{M(a),b,M(c_i)\},$$
$$M\{M\{a,M(b)\},M\{M(a),b\},c_i\}]$$

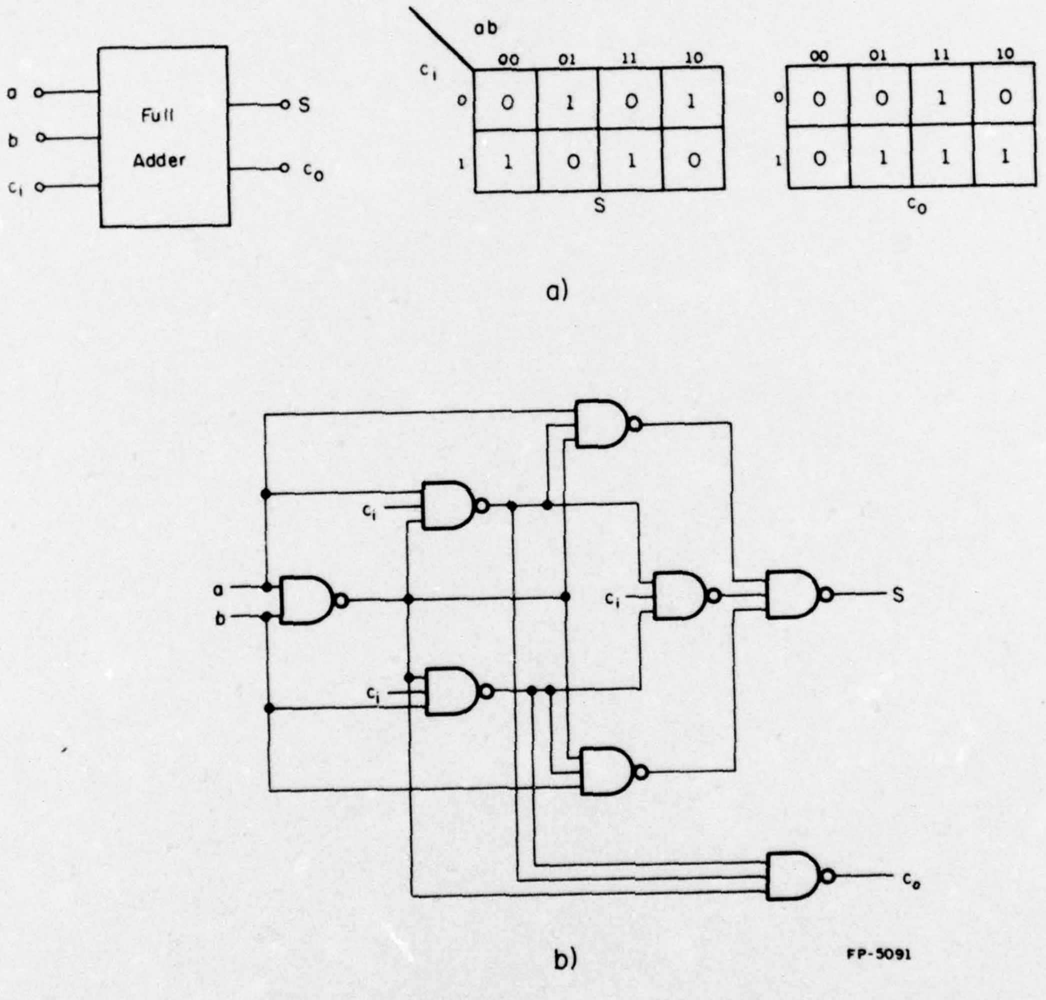


Figure 6.2. NAND/NOR algebra applied to full adder.

$$\Rightarrow S = M[M\{a,M(a,b),M(a,c_i)\},M\{M(a,b),b,M(b,c_i)\},$$
$$M\{M\{a,M(a,b)\},M\{M(a,b),b\},c_i\}].$$

Once again, using the laws of redundancy, the redundant argument M(a,b) can be inserted into both $M(a,c_i)$ and $M(b,c_i)$ within the first two major arguments, and the redundant argument c_i can be inserted into M{a,M(a,b)} and M{M(a,b),b} of the last major argument. Thus,

$$S = M[M\{a,M(a,b),M[a,M(a,b),c_i]\},M\{M(a,b),b,M[M(a,b),b,c_i]\},$$
$$M\{M[a,M(a,b),c_i],M[M(a,b),b,c_i],c_i\}].$$

Although the result looks complex, repeated appearances of the same argument make it an economical expression for S.

From the map for C_o,

$$C_o = M[M(a,b),M(b,c_i),M(a,c_i)].$$

And, using the laws of redundancy,

$$C_o = M[M(a,b),M\{M(a,b),b,c_i\},M\{a,M(a,b),c_i\}].$$

The resulting form thus has the same arguments as appear in S resulting in the circuit shown in Figure 6.2 b).

6.2. Self-Checking NAND/NOR Algebra

The NAND/NOR algebra presented permits manipulations of NAND/NOR decompositions of Boolean functions. Such decompositions are important since any Boolean function can be expressed as such a decomposition and, therefore, realized in a NAND/NOR network. However, of interest here is the realization of a function f in an alternating NAND/NOR network. If f is self-dual, then f can be expressed as a NAND/NOR decomposition and realized in a NAND/NOR network directly. If f is not self-dual, then it is dualized to form the self-dual function f_*, and f_* is expressed as a NAND/NOR decomposition and realized in a NAND/NOR network with the (0,1) clock Φ as one input. In either case the resulting NAND/NOR network is an alternating network. It is, however, not necessarily a self-checking network. To be self-checking the NAND/NOR network structure must be restricted to some degree. That restriction is essentially specified in Corollary 2.1. However, a restatement of that corollary directly as it pertains to alternating NAND/NOR networks requires some development.

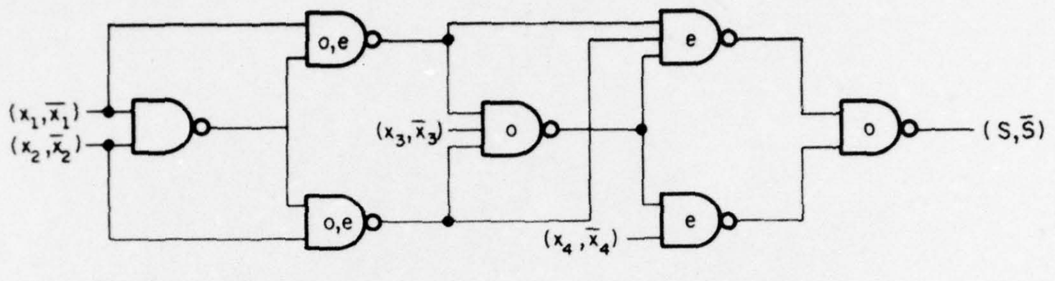
Consider an arbitrary alternating NAND/NOR network and label that network as follows. Label the output gate o, standing for odd. Label each gate which is an input to the o gate with an e, standing for even. Now, proceed toward the input, labeling each gate which is an input to an e gate with an o, and each gate which is an input to an o gate with an e. If such a labeling is possible then any output path P from a fanout gate labeled o is such that w(P) = 0, and any output path P' from a fanout gate labeled e is such that w(P') = 1. Such a network is therefore self-checking according to Corollary 2.1. If a contradiction results from the attempted labeling, then there is a fanout gate which is an input to both an odd level gate and an even level gate, and, as a result there is an output path P through the even level gate such that w(P) = 0, and an output path P' through the odd level gate such that w(P') = 1. As an example, consider the NAND/NOR network shown in Figure 6.3. The output gate is first labeled o and then labeling proceeds toward the input according to the procedure. For the network shown in a), the procedure leads to a contradiction. That network is therefore not self-checking. For the network shown in b), no contradiction is encountered, and the network is self-checking.

A design procedure which assures the production of NAND/NOR alternating networks in which fanout gates feed only odd or only even level gates is therefore desirable. One such procedure that utilizes Hohn's NAND/NOR algebra modified to carry level information meets these restrictions. The procedure starts with a two-level NAND/NOR decomposition of the self-dual function w to be realized. (Note that $w = f$ if f is self-dual, $w = f_*$ if not.) This decomposition is then subscripted to reflect level information. Then, the subscripted decomposition is transformed utilizing laws from the modified NAND/NOR algebra in an effort to obtain a more optimal multi-level realization. The decomposition obtained at each step specifies a NAND/NOR alternating network meeting the stated restrictions. To illustrate the first part of the procedure consider the function f of Example 6.1. From the map specification of f, Figure 6.1 a), the NOR decomposition,

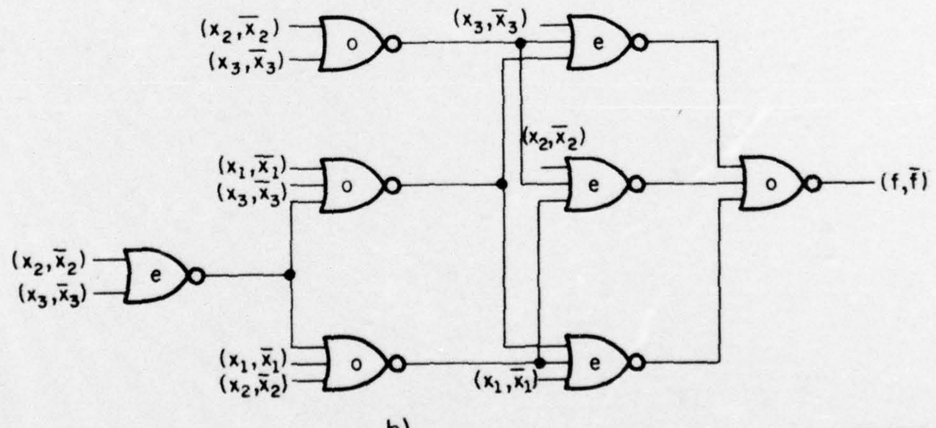
$$f = m[m(\bar{x}_1,x_2),m(\bar{x}_1,\bar{x}_3),m(x_2,\bar{x}_3,\bar{x}_4)],$$

is obtained. When this decomposition for f is subscripted to reflect level information, the subscripted decomposition,

$$f = m_o[m_e(\bar{x}_1,x_2),m_e(\bar{x}_1,\bar{x}_3),m_e(x_2,\bar{x}_3,\bar{x}_4)],$$



a)



b)

FP-5092

Figure 6.3. Level labeling of NAND/NOR networks.

results. An m_o (m_e) in this functional description translates to an odd (even) level NOR gate in the NAND/NOR network realizing f. Similarly, M_o (M_e) translates into an odd (even) level NAND gate.

Each law in Hohn's NAND/NOR algebra is modified in a similar fashion so as to carry level information. For example the hybrid associative laws become

$$M_o(f_1,f_2,\ldots,f_k,h^*) = M_o[m_e(\bar{f}_1,\bar{f}_2,\ldots,\bar{f}_k),h^*]$$

$$m_o(f_1,f_2,\ldots,f_k,h^*) = m_o[M_e(\bar{f}_1,\bar{f}_2,\ldots,\bar{f}_k),h^*]$$

if the law is applied to an odd level gate and

$$M_e(f_1,f_2,\ldots,f_k,h^*) = M_e[m_o(\bar{f}_1,\bar{f}_2,\ldots,\bar{f}_k),h^*]$$

$$m_e(f_1,f_2,\ldots,f_k,h^*) = m_e[M_o(\bar{f}_1,\bar{f}_2,\ldots,\bar{f}_k),h^*]$$

if applied to an even level gate. A simplification of notation results if L is defined as an element of the set $\{o,e\}$, $L\epsilon\{o,e\}$ where o and e are related as follows: $o = \bar{e}$, $e = \bar{o}$. The hybrid <u>associative laws</u> can then be written as,

$$M_L(f_1,f_2,\ldots,f_k,h^*) = M_L[M_{\bar{L}}(\bar{f}_1,\bar{f}_2,\ldots,\bar{f}_k),h^*]$$

$$m_L(f_1,f_2,\ldots,f_k,h^*) = m_L[M_{\bar{L}}(\bar{f}_1,\bar{f}_2,\ldots,\bar{f}_k),h^*].$$

In similar fashion the other laws or transformations are modified to form a modified NAND/NOR algebra carrying level information. The <u>commutative laws</u> become, where $i_1,i_2,\ldots,i_k$ is any permutation of $1,2,\ldots,k$,

$$M_L(f_1,f_2,\ldots,f_k) = M_L(f_{i_1},f_{i_2},\ldots,f_{i_k})$$

$$m_L(f_1,f_2,\ldots,f_k) = m_L(f_{i_1},f_{i_2},\ldots,f_{i_k}).$$

The <u>laws of factorization</u> become

$$M_L[M_{\overline{L}}(f^+,g_1^+),M_{\overline{L}}(f^+,g_2^+),\ldots,M_{\overline{L}}(f^+,g_k^+),h^*]$$

$$= M_L[M_{\overline{L}}(f^+,M_L\{M_{\overline{L}}(g_1^+),M_{\overline{L}}(g_2^+),\ldots,M_{\overline{L}}(g_k^+)\}),h^*]$$

$$m_L[m_{\overline{L}}(f^+,g_1^+),m_{\overline{L}}(f^+,g_2^+),\ldots,m_{\overline{L}}(f^+,g_k^+),h^*]$$

$$= m_L[m_{\overline{L}}(f^+,m_L\{m_{\overline{L}}(g_1^+),m_{\overline{L}}(g_2^+),\ldots,m_{\overline{L}}(g_k^+)\}),h^*].$$

The <u>laws of idempotency</u> become

$$M_L(f,f,g^*) = M_L(f,g^*)$$

$$m_L(f,f,g^*) = m_L(f,g^*).$$

The <u>laws of operation with 0 and 1</u> become

$$M_L(0,g^*) = 1 \qquad M_L(1,f^+) = M_L(f^+)$$

$$m_L(1,g^*) = 0 \qquad m_L(0,f^+) = m_L(f^+).$$

The <u>laws of complementarity</u> become

$$M_L(f,\overline{f},g^*) = 1$$

$$m_L(f,\overline{f},g^*) = 0.$$

The <u>laws of involution</u> become

$$M_L(M_{\overline{L}}(f)) = f \qquad m_L(M_{\overline{L}}(f)) = f$$

$$m_L(m_{\overline{L}}(f)) = f \qquad M_L(m_{\overline{L}}(f)) = f.$$

The <u>reduction laws</u> become

$$M_L(f,m_{\overline{L}}(f,g)) = 1$$

$$m_L(f,M_{\overline{L}}(f,g)) = 0.$$

The laws of redundancy become

$$M_L[f,M_{\bar{L}}(f,g^+),h^*] = M_L[f,M_{\bar{L}}(g^+),h^*] \qquad M_L[f,m_{\bar{L}}(\bar{f},g^+),h^*] = M_L[f,m_{\bar{L}}(g^+),h^*]$$

$$m_L[f,m_{\bar{L}}(f,g^+),h^*] = m_L[f,m_{\bar{L}}(g^+),h^*] \qquad m_L[f,M_{\bar{L}}(\bar{f},g^+),h^*] = m_L[f,M_{\bar{L}}(g^+),h^*].$$

The absorption laws become

$$M_L[M_{\bar{L}}(f^+),M_{\bar{L}}(f^+,g^*),h^*] = M_L[M_{\bar{L}}(f^+),h^*]$$

$$m_L[m_{\bar{L}}(f^+),m_{\bar{L}}(f^+,g^*),h^*] = m_L[m_{\bar{L}}(f^+),h^*].$$

The laws of consensus become

$$M_L[M_{\bar{L}}(f,g),M_{\bar{L}}(\bar{f},h),M_{\bar{L}}(g,h)] = M_L[M_{\bar{L}}(f,g),M_{\bar{L}}(\bar{f},h)]$$

$$m_L[m_{\bar{L}}(f,g),m_{\bar{L}}(\bar{f},h),m_{\bar{L}}(g,h)] = m_L[m_{\bar{L}}(f,g),m_{\bar{L}}(\bar{f},h)].$$

The laws of condensation become

$$M_L[M_{\bar{L}}(f,g^+),M_{\bar{L}}(\bar{f},g^+),h^*] = M_L[M_{\bar{L}}(g^+),h^*]$$

$$m_L[m_{\bar{L}}(f,g^+),m_{\bar{L}}(\bar{f},g^+),h^*] = m_L[m_{\bar{L}}(g^+),h^*]$$

and, identity I1 becomes

$$M_L[M_{\bar{L}}\{M_L(f_1,f_2),M_L(\bar{f}_1,\bar{f}_2)\},h^*] = M_L[M_{\bar{L}}(f_1,\bar{f}_2),M_{\bar{L}}(\bar{f}_1,f_2),h^*]$$

$$m_L[m_{\bar{L}}\{m_L(f_1,f_2),m_L(\bar{f}_1,\bar{f}_2)\},h^*] = m_L[m_{\bar{L}}(f_1,\bar{f}_2),m_{\bar{L}}(\bar{f}_1,f_2),h^*].$$

The object of applying the modified NAND/NOR algebra to the two-level subscripted NAND or NOR decomposition is to produce a multi-level decomposition containing common terms so as to economize the implementation. The full adder of Example 6.2 illustrated this approach for the unsubscripted NAND/NOR algebra. However, when the subscripted algebra is used two terms are not considered common unless they agree in subscripts as well as in type. For example suppose the subscripted algebra is applied to a two-level decomposition of a function f, and that the following multi-level decomposition is obtained,

$$f = M_o[M_e(a,b),M_e(a,c),M_e(b,c),M_e\{M_o(a,b),m_o(a,b,c)\}].$$

The subscripted terms $M_e(a,b)$ and $M_o(a,b)$ appear, but they are not considered to be common terms, and the NAND/NOR realization of f would contain a NAND gate for both $M_e(a,b)$ and $M_o(a,b)$. A form of redundancy is thus introduced automatically as required to guarantee that no NAND or NOR gate fans out to both even and odd level gates. The following example is one which results in this type of redundancy.

Example 6.3: Consider a self-dual function f specified as in Figure 6.4 a) by means of a Karnaugh map. Utilizing the prime implicates indicated, a two-level NOR decomposition can be written,

$$f = m[m(x_1,x_2,\overline{x}_3),m(x_1,\overline{x}_2,x_3),m(\overline{x}_1,\overline{x}_2,\overline{x}_3),m(\overline{x}_1,x_2,x_3)],$$

and subscripted as,

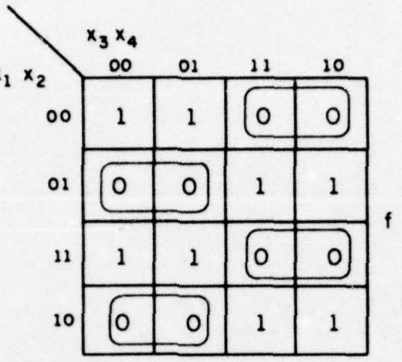
$$f = m_o[m_e(x_1,x_2,\overline{x}_3),m_e(x_1,\overline{x}_2,x_3),m_e(\overline{x}_1,\overline{x}_2,\overline{x}_3),m_e(\overline{x}_1,x_2,x_3)].$$

Then, using the factorization law, the expression,

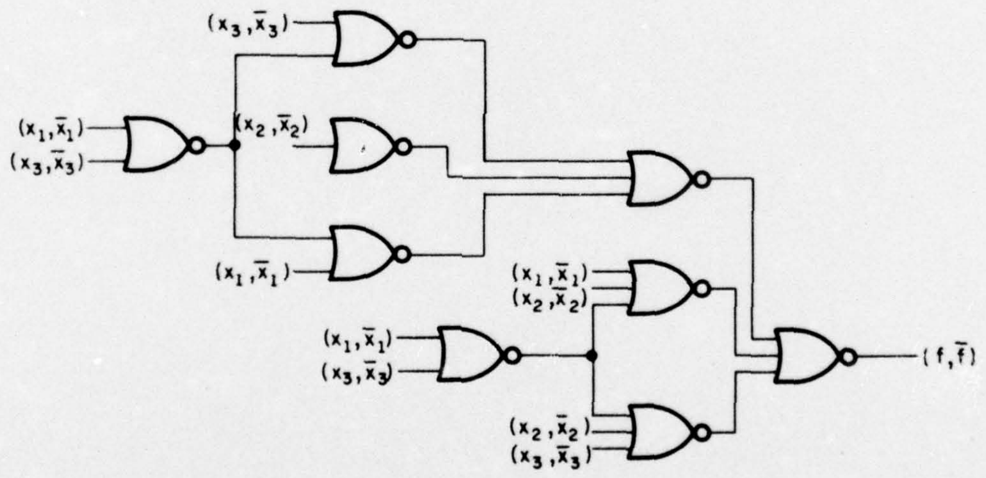
$$f = m_o[m_e\{\overline{x}_2,m_o(m_e(x_1,x_3),m_e(\overline{x}_1,\overline{x}_3))\},m_e(x_1,x_2,m_o(x_3)),m_e(x_2,x_3,m_o(x_1))],$$

is obtained. Using identity I1 on the first major term and the law of redundancy on the last two major terms yields,

$$f = m_o[m_e\{\overline{x}_2,m_o(x_1,\overline{x}_3),m_o(\overline{x}_1,x_3)\},m_e\{x_1,x_2,m_o(x_1,x_3)\},m_e\{x_2,x_3,m_o(x_1,x_3)\}].$$



a)



b)

FP-5093

Figure 6.4. Function specification and network for Example 6.3.

Eliminating complements using the law of redundancy then yields,

$$f = m_o[m_e\{m_o(x_2), m_o(x_1, m_e(x_1, x_3)), m_o(x_3, m_e(x_1, x_3))\},$$
$$m_e\{x_1, x_2, m_o(x_1, x_3)\}, m_e\{x_2, x_3, m_o(x_1, x_3)\}].$$

Based on this multi-level decomposition, a NOR network containing an even level gate for $m_e(x_1, x_3)$ and an odd level gate for $m_o(x_1, x_3)$ is constructed. This network is shown in Figure 6.4 b).

The procedure described can be applied to functions which are not self-dual also. The result is, of course, not an alternating network, but simply a NAND/NOR network having the property that no NAND or NOR gate feeds both even and odd level gates. The following example illustrates.

Example 6.4: The function f has a Karnaugh map specification shown in Figure 6.5 a). From that specification the two-level NAND decomposition,

$$f = M[M(c, \bar{a}, \bar{b}), M(\bar{c}, a, b)],$$

can be written utilizing the prime implicants shown. Subscripted, the decomposition becomes

$$f = M_o[M_e(c, \bar{a}, \bar{b}), M_e(\bar{c}, a, b)]$$

or,

$$f = M_o[M_e(c, M_o(b), M_o(a)), M_e(M_o(c), a, b)].$$

Repeated use of the law of redundancy then results in the following sequence of transformations,

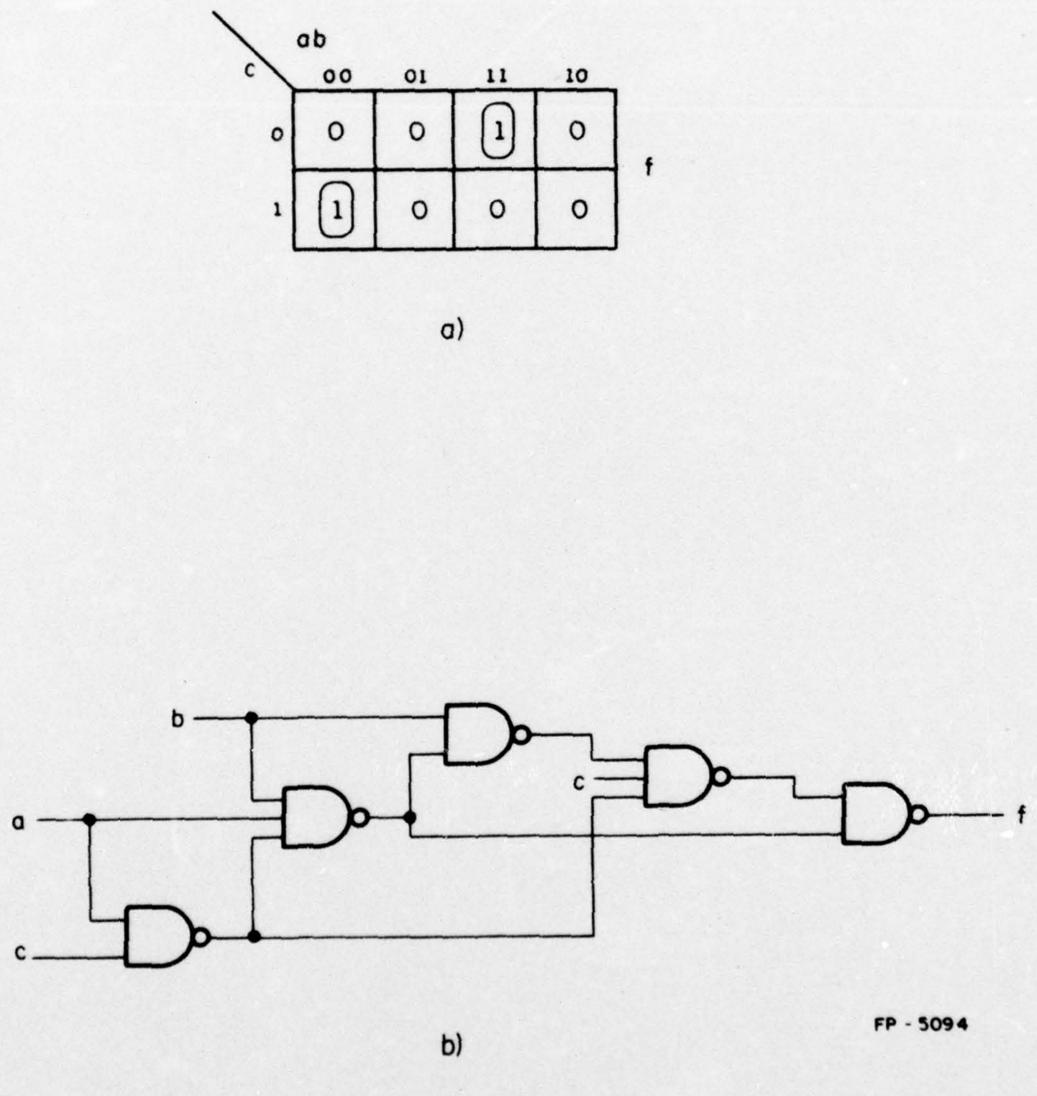


Figure 6.5. Function specification and network for Example 6.4.

$$
\begin{aligned}
f &= M_o[M_e\{c,M_o(b,M_e(a)),M_o(a)\},M_e\{a,b,M_o(a,c)\}],\\
&= M_o[M_e\{c,M_o(b,M_e(a)),M_o(a,c)\},M_e\{a,b,M_o(a,c)\}]\\
&= M_o[M_e\{c,M_o(b,M_e(a,c),M_e(a)),M_o(a,c)\},M_e\{a,b,M_o(a,c)\}]\\
&= M_o[M_e\{c,M_o(b,M_e(a,c),M_e(a,M_o(a,c))),M_o(a,c)\}, M_e\{a,b,M_o(a,c)\}]\\
&= M_o[M_e\{c,M_o(b,M_e(a,b,M_o(a,c))),M_o(a,c)\},M_e\{a,b,M_o(a,c)\}].
\end{aligned}
$$

The resulting NAND network, shown in Figure 6.5 b), has no gate which feeds both even and odd level gates.

7. COMPLETE SETS OF ALTERNATING LOGIC PRIMITIVES

7.1. Weak Complete Alternating Primitive Sets

Consider a combinational network N realizing an arbitrary function f with n inputs $x_1,x_2,\ldots,x_n$ and one output y. The methods described previously for realizing N as an alternating circuit utilized a dualization procedure for functions f which were not self-dual so as to produce a self-dual functional specification representing the original functional specification. Such a dualization utilized a system clock as an input in such a manner that with the clock at logical '0' and $x_1,x_2,\ldots,x_n$ applied as inputs, $f(x_1,x_2,\ldots,x_n)$ was produced. With the clock at logical '1' and $\bar{x}_1,\bar{x}_2,\ldots,\bar{x}_n$ applied as input, $\bar{f}(x_1,x_2,\ldots,x_n)$ was produced. For a function f which was already self-dual, the functional specification representing f was the functional specification for f itself, and no clock input was required. Once these representative functional specifications were determined, they were realized using conventional logic primitives, e.g. AND/OR/NOT, NAND, or NOR, in networks designed to be self-checking for all single faults. In each case, design procedures using the conventional sets of logic primitives led to networks of restricted structure.

Now, consider a set F of self-dual functions $F = \{f^{(1)},f^{(2)},\ldots,f^{(p)}\}$ and suppose that this set of logic primitives is to be used to realize the function f directly. That is, the function f is to be realized as a composition of functions from the set F. A composition of self-dual functions, Ibuki [20], is itself self-dual. Thus, while the realization of any composition of functions from the set F is an alternating circuit, only alternating circuits realizing self-dual functions result from the compositions. In alternating logic design, however, the clock is available and is itself a self-dual function representing the conventional o function. So, every Boolean function can be represented in an alternating circuit constructed as a composition of alternating primitives $f^{(1)},f^{(2)},\ldots,f^{(p)}$ if and only if $F' = \{f^{(1)},f^{(2)},\ldots,f^{(p)},o\}$ is a complete set of logical primitives in conventional terms. If this is true, then $\{f^{(1)},f^{(2)},\ldots,f^{(p)}\}$ will be called a weak complete set of alternating logic primitives. To determine what restrictions are placed on $f^{(1)},f^{(2)},\ldots,f^{(p)}$ such that $\{f^{(1)},f^{(2)},\ldots,f^{(p)},o\}$ is a complete set of logic primitives in the conventional sense requires some background and development in the theory of complete sets of logic primitives.

Definition 7.1: The function set S generated by a function set R = $\{f_1,f_2,\ldots,f_q\}$ is the set of functions realizable as a composition of functions from the set R and is denoted by [R].

Obviously, the set R is a complete set if for every Boolean function $f, f \in [R]$. Further, if K^N denotes the set of all Boolean functions of N or fewer variables and $[R] = K^N$, then R is complete. This follows from the fact that if a function set is complete for a certain value of N, $N \geq 2$, then the function set is complete for any value of N, [20]. A minimal complete function set is a complete set which fails to be complete if any one primitive is removed from it. A maximal incomplete set of logic primitives is a set of logic functions which becomes complete if there is added to it any other function which is not an element of the set.

Five functional properties are instrumental in determining whether or not a set of logic primitives is complete. They are the 0-preserving, 1-preserving, self-dual, linear, and monotonic properties of the functions in the set R.

Definition 7.2: A function f is 0-preserving if the value of the function is 0 when every input variable takes on a value of 0.

$$f(0,0,\ldots,0) = 0$$

Let M_1 designate the set of all 0-preserving functions.

Definition 7.3: A function f is 1-preserving if the value of the function is 1 when every input variable takes on the value 1.

$$f(1,1,\ldots,1) = 1$$

Let M_2 designate the set of all 1-preserving functions.

Definition 7.4: A function f is self-dual iff the function f and its dual function f^D are identical or equivalently,

$$f(x_1,x_2,\ldots,x_n) = \overline{f}(\overline{x}_1,\overline{x}_2,\ldots,\overline{x}_n).$$

Let M_3 designate the set of all self-dual functions.

Definition 7.5: A function f is linear if it can be written in the form

$$f(x_1,x_2,\ldots,x_n) = g_o \oplus (g_1x_1 \oplus g_2x_2 \oplus \cdots \oplus g_nx_n).$$

Let M_4 designate the set of all linear functions.

<u>Definition 7.6</u>: A function f is <u>monotonically increasing</u> if for every $x_{11},x_{12},\ldots,x_{1n} \geq x_{21},x_{22},\ldots,x_{2n}$ then

$$f(x_{11},x_{12},\ldots,x_{1n}) \geq f(x_{21},x_{22},\ldots,x_{2n}).$$

Let M_5 designate the set of all monotonically increasing functions.

Now, consider again the set of logic primitives $F' = \{f^{(1)},f^{(2)},\ldots,f^{(p)},o\}$ where $f^{(1)},f^{(2)},\ldots,f^{(p)}$ are self-dual functions. Completeness of the set F' is dependent on the properties of the self-dual functions $f^{(1)},f^{(2)},\ldots,f^{(p)}$. These dependencies are defined in the following.

<u>Lemma 7.1</u>: If $F' \not\subset M_1$, then $\bar{x}\epsilon[F']$ and $1\epsilon[F']$.

<u>Proof</u>: If $F' \not\subset M_1$, then there exists a function $f\epsilon F'$ which is not 0-preserving. So then, either $f\epsilon M_2 \cap \bar{M}_1$, or $f\epsilon \bar{M}_2 \cap \bar{M}_1$. But, *if f is self-dual and not* 0-preserving then

$$f(0,0,\ldots,0) = 1$$
$$\Rightarrow f(1,1,\ldots,1) = 0,$$

and f is not 1-preserving. So, $f \not\epsilon M_2 \cap \bar{M}_1$. That is, $f\epsilon \bar{M}_2 \cap \bar{M}_1$. But then $\bar{x}\epsilon[F']$, since the function $\bar{x}$ can be obtained from f by tying all inputs of f together. Thus, for 0 in, $x_1 = 0$, $x_2 = 0,\ldots,x_n = 0$ and, since f is not 0-preserving, a 1 is produced as an output. For 1 in, $x_1 = 1, x_2 = 1,\ldots,x_n = 1$ and a 0 is produced as f is not 1-preserving. Now, since $\bar{x}\epsilon[F']$ and $0\epsilon[F']$, $1\epsilon[F']$. Q.E.D.

Further, Ibuki [20] has proved the following lemma. The proof appearing here is a version of that proof.

Lemma 7.2: If $R \not\subset M_4$ and $\bar{x}, 0, 1 \epsilon [R]$ then R is a complete set of primitives, where R is an arbitrary set of functions not necessarily self-dual.

Proof: If $R \not\subset M_4$ then there exists a function $f \epsilon R$ which is nonlinear. That is, if f is expanded in the form

$$f = g_o \oplus (g_1 x_1 \oplus g_2 x_2 \oplus \ldots \oplus g_n x_n) \oplus (g_{12} x_1 x_2 \oplus g_{13} x_1 x_3 \oplus \ldots \oplus g_{1n} x_1 x_n \oplus g_{23} x_2 x_3 \oplus \ldots$$

$$\oplus g_{n-1} g_n x_{n-1} x_n) \oplus \ldots \oplus g_{12\ldots n} x_1 x_2 \ldots x_n$$

there exists at least one term in the expansion of order t where $2 \leq t \leq n$. Let s be the order of the term of minimum order in that range. Let y_1 denote the variable with the smallest subscript in the term, y_s denote the variable with the next smallest subscript,..., and y_s denote the variable with the largest subscript in the term. Now, if $y_3, y_4, \ldots$, and y_s are constrained to be 1 and all remaining variables except y_1 and y_2 are constrained to be 0, then a new 2-variable function f_1 results. Certainly, $f_1 \epsilon [R]$ by construction.

$$f_1 = \alpha_o \oplus \alpha_1 y_1 \oplus \alpha_2 y_2 \oplus y_1 y_2$$

where $\alpha_0, \alpha_1, \alpha_2 \epsilon \{0,1\}$. Now, $\alpha_1 \oplus x_2 \epsilon [R]$ for $\alpha_1 = 0$ or 1 since $\bar{x} \epsilon f$. Similarly

$$\alpha_2 \oplus x_1 \epsilon [R].$$

So,

$$f_2 = \alpha_o \oplus \alpha_1 (\alpha_2 \oplus x_1) \oplus \alpha_2 (\alpha_1 \oplus x_2) \oplus (\alpha_2 \oplus x_1)(\alpha_1 \oplus x_2) \epsilon [R]$$

$$\Rightarrow f_2 = \alpha_o \oplus \alpha_1 \alpha_2 \oplus x_1 x_2.$$

But then,

$$f_3 = \alpha_o \oplus \alpha_1 \alpha_2 \oplus f_2 \epsilon [R]$$

$$\Rightarrow f_3 = x_1 x_2 \epsilon [R].$$

Similarly, $f_4 = f_3\{\overline{f_3(x_1,x_2)},\overline{f_3(\overline{x}_1,\overline{x}_2)}\}\epsilon[R]$

$$\Rightarrow f_4 = \overline{x_1x_2}\cdot\overline{\overline{x}_1\overline{x}_2}\epsilon[R]$$

$$\Rightarrow f_4 = x_1 \oplus x_2 \;\epsilon\; [R].$$

But then, R must be complete since $\{x_1x_2, x_1 \oplus x_2, 1\}$ is a complete set. Q.E.D.

Theorem 7.1: A set $F = \{f^{(1)}, f^{(2)}, \ldots, f^{(p)}\}$ of self-dual functions is a weak complete set of alternating logic primitives if $f^{(i)}$ is not 0-preserving for some i, $1 \leq i \leq p$, and $f^{(j)}$ is nonlinear for some j, $1 \leq j \leq p$ where possibly $i = j$.

Proof: The set F is a weak complete set of alternating logic primitives if the set $F' = \{f^{(1)}, f^{(2)}, \ldots, f^{(p)}, 0\}$ is a complete set in the conventional sense. And by Lemma 7.1 and Lemma 7.2, F' is complete if $F' \not\subset M_1$ and $F' \not\subset M_4$, or in other words if $f^{(i)}$ is not 0-preserving for some i, $1 \leq i \leq p$ and $f^{(j)}$ is nonlinear for some j, $1 \leq j \leq p$. Q.E.D.

Corollary 7.1: The set $F = \{f^{(1)}, f^{(2)}, \ldots, f^{(p)}\}$ of self-dual functions is a minimal weak complete set of alternating logic primitives if for every proper subset G of F, $G \subset M_1$ or $G \subset M_4$.

Corollary 7.2: Any minimal weak complete set of alternating logic primitives contains at most 2 self-dual logic primitives.

7.2. Strong Complete Alternating Primitive Sets

In any application of the theory outlined thus far it is implicitly assumed that the (0,1) clock, the 0-function in an alternating network, may be used as an input to any variable line in a network application of any function from the set F. This means that the (0,1) clock may be input on 2 or more variable lines in an application of a given logic primitive in the composition of a desired alternating network from the set F of alternating logic primitives. This, however, may result in an undesirable loading of the clock. If application of the clock is limited to at most one input line of any logic primitive, a somewhat different theory of alternating logic primitives results.

Once again let $F = \{f^{(1)}, f^{(2)}, \ldots, f^{(p)}\}$ where $f^{(j)}$ is a function of k_j variables, $1 \leq j \leq p$, be a set of self-dual functions. The set F is a strong complete set of alternating logic primitives if an alternating network representing any Boolean function can be constructed by composing elements of F with the provision that the (0,1) clock is used as an input to at most one variable of any given alternating logic primitive. Before the problem can be restated in terms pertinent to conventional theory of complete sets, the concept of a 0-subfunction must be defined.

Definition 7.7: The i-th 0-subfunction $f_{0,i}$ of a function $f(x_1,x_2,\ldots,x_n)$ of n variables is the function obtained by constraining the i-th variable in $f(x_1,x_2,\ldots,x_n)$ to a 0. That is,

$$f_{0,i}(x_1,x_2,\ldots,x_{i-1},x_{i+1},\ldots,x_n) = f(x_1,x_2,\ldots,x_{i-1},0,x_{i+1},\ldots,x_n).$$

For a function $f^{(j)}$, $1 \leq j \leq p$, from the set F the i-th 0-subfunction will be denoted $f_{0,i}^{(j)}$. Now, the set $F = \{f^{(1)},f^{(2)},\ldots,f^{(p)}\}$ will be a strong complete set of alternating logic primitives if

$$T = \{f^{(1)},f^{(2)},\ldots,f^{(p)}, \bigcup_{m=1}^{p} \bigcup_{i=1}^{k_m} f_{0,i}^{(m)}\}$$

is a set of complete logic primitives in the conventional sense. Further, the completeness of T can be determined using the following theorem, Ibuki [20].

Theorem 7.2: A function set R is a complete set of functions if $F \not\subset M_i$ for $1 \leq i \leq 5$.

The proof of the theorem is presented in [20] and is not repeated here. Now, the number of functions in T is at most $p + \sum_{m=1}^{p} k_m$ as some of the 0-subfunctions may be identical. At any rate it appears that each function in T must be classified according to the five properties, 0-preserving, 1-preserving, self-dual, linear, and monotonic so as to determine if T is a complete set. Fortunately, this is not the case as will be shown.

Lemma 7.3: No 0-subfunction $f_{0,i}$ of a self-dual function $f(x_1,x_2,\ldots,x_n)$ is self-dual, $1 \leq i \leq n$.

Proof: By hypothesis f is a self-dual function of n variables. By way of contradiction assume that $f_{0,i}$ is self-dual. Then,

$$f(x_1,x_2,\ldots,x_{i-1},0,x_{i+1},\ldots,x_n) = \overline{f}(\overline{x}_1,\overline{x}_2,\ldots,\overline{x}_{i-1},0,\overline{x}_{i+1},\ldots,\overline{x}_n).$$

But, since f is self-dual

$$f(x_1,x_2,\ldots,x_{i-1},1,x_{i+1},\ldots,x_n) = \bar{f}(\bar{x}_1,\bar{x}_2,\ldots,\bar{x}_{i-1},0,\bar{x}_{i+1},\ldots,\bar{x}_n)$$

$$\Rightarrow\ f(x_1,x_2,\ldots,x_{i-1},0,x_{i+1},\ldots,x_n) = f(x_1,x_2,\ldots,x_{i-1},1,x_{i+1},\ldots,x_n)$$

for every $x_1,x_2,\ldots,x_{i-1},x_{i+1},\ldots,x_n$ and it therefore follows that f is a function of n-1 variables, a contradiction. Q.E.D.

<u>Lemma 7.4</u>: If $f(x_1,x_2,\ldots,x_n)$ is a linear function of n variables, then every 0-subfunction $f_{0,i}$, $1 \leq i \leq n$, is linear.

<u>Proof</u>: By hypothesis f is a linear function and can therefore be written in the form

$$f = g_o \oplus (g_1x_1 \oplus g_2x_2 \oplus \ldots \oplus g_nx_n)$$

where $g_o,g_1,\ldots,g_n \epsilon \{0,1\}$. But then,

$$f_{0,i}(x_1,x_2,\ldots,x_{i-1},x_{i+1},\ldots,x_n) = g_o \oplus (g_1x_1 \oplus \ldots \oplus g_{i-1}x_{i-1} \oplus g_{i+1}x_{i+1} \oplus \ldots \oplus g_nx_n).$$

The function $f_{0,i}$ is therefore a linear function also. Q.E.D.

<u>Lemma 7.5</u>: If $f(x_1,x_2,\ldots,x_n)$ is a monotonically increasing function of n variables, then every 0-subfunction, $f_{0,i}$, $1 \leq i \leq n$, is monotonically increasing.

<u>Proof</u>: By way of contradiction assume that $f(x_1,x_2,\ldots,x_n)$ is monotonically increasing but that $f_{0,i}$ for some i, $1 \leq i \leq n$, is not. Then, there exists a pair of (n-1)-tuples

$$x_{1,1}x_{2,1}\ldots x_{i-1,1}\ x_{i+1,1}\ldots x_{n,1} \geq x_{1,2}x_{2,2}\ldots x_{i-1,2}\ x_{i+1,2}\ldots x_{n,2}$$

for which

$$f_{0,i}(x_{1,1},x_{2,1},\ldots,x_{i-1,1},x_{i+1,1},\ldots,x_{n,1}) < f_{0,i}(x_{1,2},x_{2,2},\ldots,x_{i-1,2},$$
$$x_{i+1,2},\ldots,x_{n,2}).$$

But, since

$$f_{0,i}(x_{1,1},x_{2,1},\ldots,x_{i-1,1},x_{i+1,1},\ldots,x_{n,1}) = f(x_{1,1},x_{2,1},\ldots,x_{i-1,1},0,$$
$$x_{i+1,1},\ldots,x_{n,1})$$

and

$$f_{0,i}(x_{1,2},x_{2,2},\ldots,x_{i-1,2},x_{i+1,2},\ldots,x_{n,2}) = f(x_{1,2},x_{2,2},\ldots,x_{i-1,2},0,$$
$$x_{i+1,2},\ldots,x_{n,2})$$

for

$$x_{1,1}x_{2,1}\cdots x_{i-1,1}\ x_{i+1,1}\cdots x_{n,1} \geq x_{1,2}x_{2,2}\cdots x_{i-1,2}\ x_{i+1,2}\cdots x_{n,2},$$

$$f(x_{1,1},x_{2,1},\ldots,x_{i-1,1},0,x_{i+1,1},\ldots,x_{n,1}) < f(x_{1,2},x_{2,2},\ldots,x_{i-1,2},0,$$
$$x_{i+1,2},\ldots,x_{n,2}).$$

The function f is therefore not monotonically increasing, a contradiction.

Q.E.D.

The above lemmas have thus shown that the set

$$T = \{f^{(1)},f^{(2)},\ldots,f^{(p)},\bigcup_{m=1}^{p}\ \bigcup_{i=1}^{k_m} f_{0,i}^{(m)}\}$$

always contains a function which is not self-dual. Specifically every subfunction $f_{0,i}$ is not self-dual. Further, the lemmas show that to determine if T is a subset of the set of linear functions, $T \subset M_4$, or if T is a subset of the set of monotonically increasing functions, $T \subset M_5$, only the functions $f^{(1)},f^{(2)},\ldots,f^{(p)}$ need be examined with respect to these properties.

Theorem 7.3: A set of self-dual functions $F = \{f^{(1)}, f^{(2)}, \ldots, f^{(p)}\}$ is a strong complete set of alternating logic primitives if $f^{(i)}$ is not 0-preserving for some i, $1 \leq i \leq p$, $f^{(j)}$ is not linear for some j, $1 \leq j \leq p$, and $f^{(k)}$ is not monotonically increasing for some k, $1 \leq k \leq p$.

Proof: The set $F = \{f^{(1)}, f^{(2)}, \ldots, f^{(p)}\}$ is a strong complete set of alternating logic primitives if

$$T = \{f^{(1)}, f^{(2)}, \ldots, f^{(p)}, \bigcup_{m=1}^{p} \bigcup_{i=1}^{k_m} f_{0,i}^{(m)}\}$$

is complete in the conventional sense. And, by Theorem 7.2, T is complete if $T \not\subset M_i$ for $1 \leq i \leq 5$. Now, F contains a function $f^{(\ell)}$, $1 \leq \ell \leq p$, which is not 0-preserving by hypothesis. So, $T \not\subset M_1$. But if a self-dual function is not 0-preserving it is not 1-preserving. So, $T \not\subset M_2$. By Lemma 7.3, $T \not\subset M_3$ as every subfunction $f_{0,i}$ is not self-dual. Finally, by Lemma 7.4 and Lemma 7.5 $T \not\subset M_4$ and $T \not\subset M_5$ if $F \not\subset M_4$ and $F \not\subset M_5$. So, T is complete if $F \not\subset M_1, M_4, M_5$. Q.E.D.

Corollary 7.3: A proper set of self-dual functions $F = \{f^{(1)}, \ldots, f^{(p)}\}$ is a minimal strong complete set of alternating logic primitives if for every proper subset G of F, $G \subset F$, then $G \subset M_i$ for some i, $i = 1,4,5$.

Theorem 7.4: Any weak complete set of alternating logic primitives is also a strong complete set of alternating logic primitives.

Proof: Assume a set $F = \{f^{(1)}, f^{(2)}, \ldots, f^{(p)}\}$ is a weak complete set of alternating logic primitives. Then $F \not\subset M_1, M_4$. But, if $F \not\subset M_1$, then there is a function $f^{(i)}$, $1 \leq i \leq p$, $f^{(i)} \in F$, having the property that $f^{(i)}$ is not 0-preserving. So,

$$f(0,0,\ldots,0) = 1$$

and by self-duality

$$f(1,1,\ldots,1) = 0.$$

But then there is a pair of n-tuples namely 000...0 and 111...1 with the following relationship

$$111\ldots1 > 000\ldots0$$

$$f(1,1,\ldots,1) = 0 < 1 = f(0,\ldots,0).$$

So, f is not monotonically increasing, and $F \not\subset M_5$. So, if $F \not\subset M_1,M_4$ then $F \not\subset M_1,M_4,M_5$. Q.E.D.

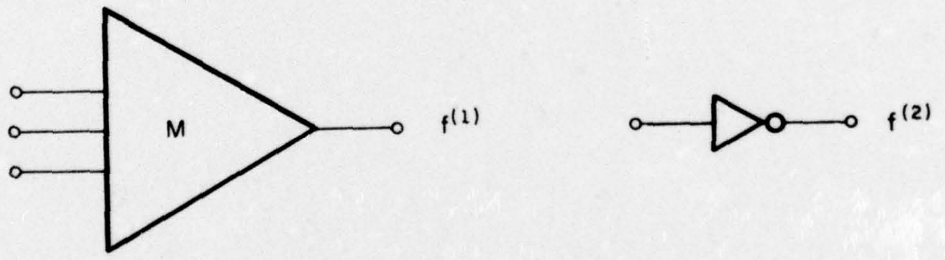
7.3. Alternating Design with the Majority Module

Necessary conditions that a set of p alternating logic primitives $F = \{f^{(1)},f^{(2)},\ldots,f^{(p)}\}$ be a complete set of alternating logic primitives have been derived. Application of these conditions to minimal complete sets resulted in a bound on p, $p \leq 2$. So, consider the set $F = \{f^{(1)},f^{(2)}\}$ in this regard where $f^{(1)}$ and $f^{(2)}$ are functions specified in Figure 7.1. Both $f^{(1)}$ and $f^{(2)}$ are self-dual, and $f^{(1)} \not\subset M_4$, $f^{(2)} \not\subset M_1$ so F = $F = \{f^{(1)},f^{(2)}\} \not\subset M_1,M_4$. The set F is therefore a complete set of alternating logic primitives. Further, since $f^{(1)} \subset M_1$ and $f^{(2)} \subset M_4$ the set F is a minimal complete set of alternating logic primitives.

Obviously, the module $f^{(2)}$ is the standard NOT gate or module. The module $f^{(1)}$, however, is not a standard gate or module. The module $f^{(1)}$ is a majority gate, namely a 3-input majority gate. Since F is complete, with a basic module realizing $f^{(1)}$ and a basic module realizing

$$f^{(1)}: \quad \begin{array}{c|cccc} x_1 \backslash x_2 x_3 & 00 & 01 & 11 & 10 \\ \hline 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 1 & 1 \end{array}$$

$$f^{(2)}: \quad \begin{array}{cc} \multicolumn{2}{c}{x} \\ 0 & 1 \\ \hline 1 & 0 \end{array}$$



FP-5095

Figure 7.1. M module and NOT module.

$f^{(2)}$, alternating circuits representing any Boolean function can be constructed. But what design procedures define the required construction? Some insight into such a design procedure might be gained by examining the functions represented by $f^{(1)}$ when the (0,1) clock is applied as input first on x_1, then on x_2, and finally on x_3. These functions are $f^{(1)}_{0,1}$, $f^{(1)}_{0,2}$, and $f^{(1)}_{0,3}$ respectively.

$$f^{(1)} = x_1x_3 + x_1x_2 + x_2x_3$$

$$f^{(1)}_{0,1} = x_2x_3 \qquad f^{(1)}_{0,2} = x_1x_3 \qquad f^{(1)}_{0,3} = x_1x_2$$

Thus, with the (0,1) clock on any input to the M module, the M module represents the AND of the other inputs. The M module configured in this manner will be called the "alternating AND."

Now, since the NOT gate, $f^{(2)}$, is in the set F, the (0,1) clock can be applied as input to that gate so as to produce a complemented clock, the (1,0) clock. When this clock is applied to the M module as input, first on x_1, then on x_2, and finally on x_3, an alternating circuit representing $f^{(1)}_{1,1}$, $f^{(1)}_{1,2}$, and $f^{(1)}_{1,3}$, respectively, results.

$$f^{(1)} = x_1x_3 + x_1x_2 + x_2x_3$$

$$f^{(1)}_{1,1} = x_2 + x_3 \qquad f^{(1)}_{1,2} = x_1 + x_3 \qquad f^{(1)}_{1,3} = x_1 + x_2$$

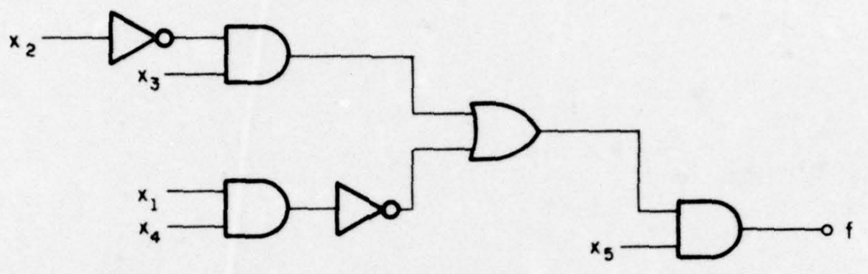
Thus, with the (1,0) clock on any input to the M module, the M module represents the OR of the other inputs. The M module configured in this manner will be called the "alternating OR."

A design proceudre therefore presents itself. Assume a Boolean function f is to be represented in an alternating system. A conventional AND-OR-NOT realization of f is first found utilizing standard design techniques. Then, every AND gate is replaced by an alternating AND. Similarly, every OR gate is replaced by an alternating OR. Every NOT gate remains intact. The additional (1,0) clock is obtained from the (0,1) clock using an additional NOT gate. The resulting network is certainly an alternating network since it constitutes a composition of alternating logic primitives, namely $f^{(1)}$ and $f^{(2)}$. Primary inputs to the conventional AND-OR-NOT realization become primary inputs to the alternating circuit and in the case of the alternating system are assumed to alternate in synchronism with the (0,1) clock. Further, the resulting alternating network must represent f as each conventional gate is replaced with an equivalent alternating gate.

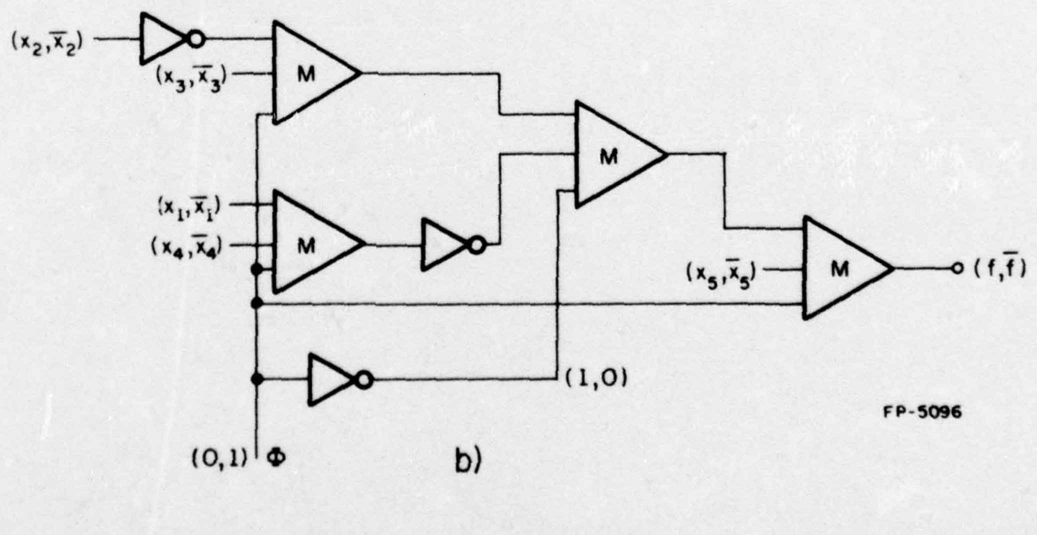
Example 7.1: Assume that a function f is to be realized in an alternating logic design where

$$f = (\overline{x}_2 x_3 + \overline{x_1 x_4}) x_5 .$$

To generate the desired alternating network a conventional AND/OR/NOT design is first generated directly from the functional description for f. This network is shown in Figure 7.2 a). Then each AND gate is replaced by an alternating AND and each OR by an alternating OR. The (0,1) clock required by the alternating AND's is assumed to be available. The (1,0) clock required by the alternating OR's is obtained from the (0,1) clock using an additional NOT gate. The alternating network is shown in Figure 7.2 b).



a)



b)

Figure 7.2. Example majority logic design.

7.4. Alternating Design with the Minority Module

Consider the set $F = \{f^{(1)}\}$, where $f^{(1)}$ is specified as in Figure 7.3. Now, $f^{(1)} \not\subset M_1$, $f^{(1)} \not\subset M_4$ so F is a complete set of alternating logic primitives. That is, the function $f^{(1)}$ is complete in itself being self-dual, but neither 0-preserving nor linear. If the (0,1) clock is applied on any input of the $\overline{M}$ module realizing $f^{(1)}$, then in an alternating network, the NAND of the other two inputs is realized.

$$f^{(1)} = \overline{(x_1+x_2)(x_1+x_3)(x_2+x_3)}$$

$$f^{(1)}_{0,1} = \overline{x_2x_3} \qquad f^{(1)}_{0,2} = \overline{x_1x_3} \qquad f^{(1)}_{0,3} = \overline{x_1x_2}$$

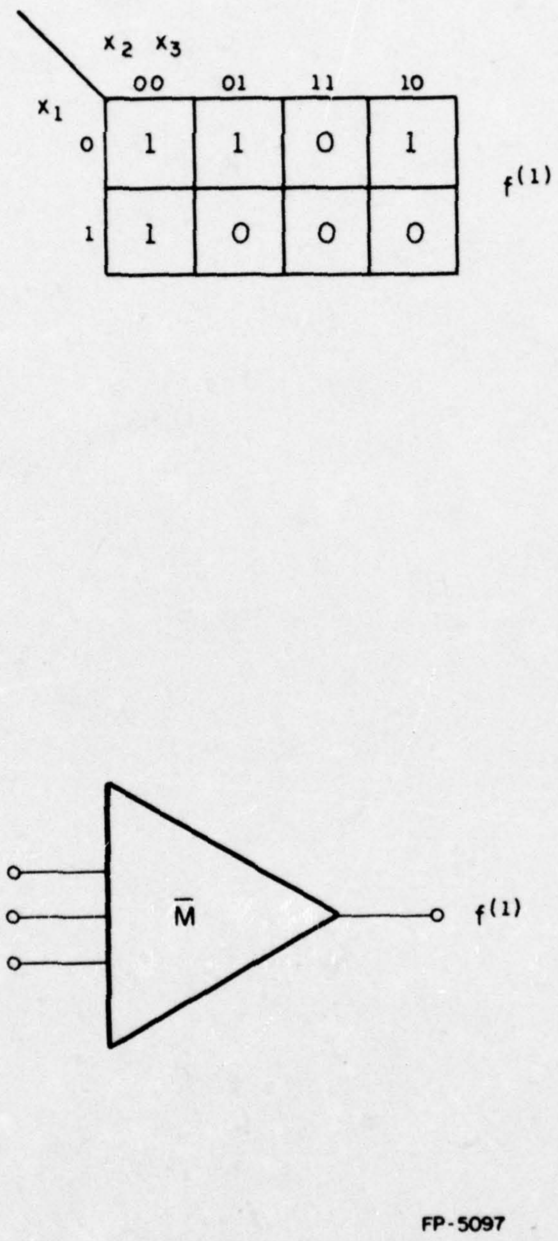
The $\overline{M}$ module configured with the (0,1) clock as one input will be referred to as the "alternating NAND." Similarly, if the (1,0) clock is applied on any input of the $\overline{M}$ module, then in an alternating network the NOR of the other two inputs is realized.

$$f^{(1)} = \overline{(x_1+x_2)(x_1+x_3)(x_2+x_3)}$$

$$f^{(1)}_{1,1} = \overline{x_2+x_3} \qquad f^{(1)}_{1,2} = \overline{x_1+x_3} \qquad f^{(1)}_{1,3} = \overline{x_1+x_2}$$

The $\overline{M}$ module configured with the (1,0) clock as one input will be referred to as the "alternating NOR."

One design procedure for obtaining alternating networks composed of minority gates thus proceeds as follows. The Boolean function to be realized in an alternating network is first realized as a conventional NAND network. Then, each NAND gate in this network is replaced by an alternating NAND to form an alternating network. Primary network inputs in the conventional design become alternating primary inputs in the alternating



$x_1 \backslash x_2 x_3$	00	01	11	10
0	1	1	0	1
1	1	0	0	0

Figure 7.3. Minority module, $\overline{M}$.

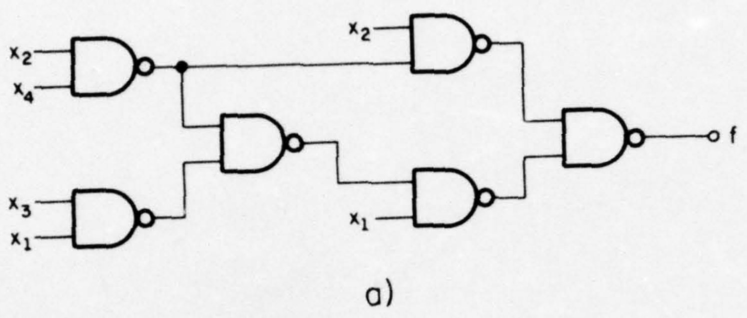
design, alternating in synchronism with the (0,1) clock, which is used as input to each alternating NAND. The result is an alternating circuit realizing f.

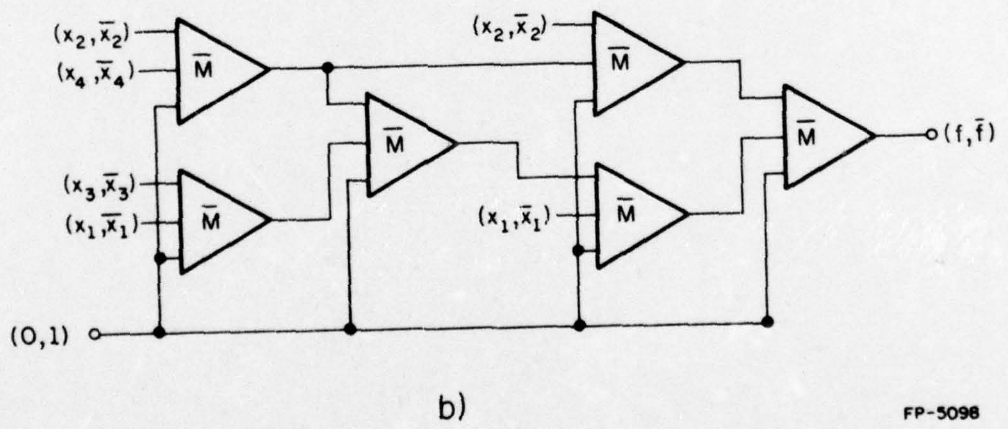
<u>Example 7.2</u>: Assume that a function f is to be realized in an alternating logic design using minority gates where f is given as

$$f = \overline{\overline{(x_2\ \overline{x_2x_4})}\,\overline{(x_1\ \overline{\overline{x_2x_4}\ \overline{x_1x_3}})}}.$$

Since f is already specified as a NAND decomposition, a conventional NAND network realizing f can be obtained from the functional specification directly. This network is shown in Figure 7.4 a). To obtain an alternating network f each NAND in the conventional network is first replaced with an $\overline{M}$ module having one input tied to the (0,1) clock which is assumed to be available. Primary inputs to the conventional network become synchronized alternating primary inputs to the alternating network. The resulting network is an alternating network realizing the function f and is shown in Figure 7.4 b).

The minority logic alternating network of least cost is not necessarily obtained by the above procedure even if the conventional design from which the alternating design is constructed is optimal. The reason for this is that no attempt is made to use the $f^{(1)}$ module, independent of the (0,1) clock as a basic gate. The following example illustrates the problem.





FP-5098

Figure 7.4. Example minority logic design.

Example 7.3: Consider the function f given as

$$f = \overline{\overline{(x_4 \overline{\overline{(\overline{x_1x_2})(\overline{x_1x_3})(\overline{x_2x_3})}})}\ \overline{x_1x_3}}\ .$$

A conventional NOR network realizing f would utilize 8 NOR gates. The alternating network obtained from this realization by the minority logic alternating design procedure would therefore utilize 8 $\overline{M}$-modules. However, a realization using only 4 $\overline{M}$-modules exists. This realization is obtained as follows. Rewrite $f^{(1)}$ in its dual form,

$$f^{(1)} = \overline{(x_1+x_2)(x_2+x_3)(x_1+x_3)}$$

$$\Rightarrow f^{(1)} = \overline{x_1x_2 + x_2x_3 + x_1x_3},$$

and then formulate it as a NAND decomposition,

$$f^{(1)} = \overline{\overline{(\overline{x_1x_2})(\overline{x_2x_3})(\overline{x_1x_3})}}.$$

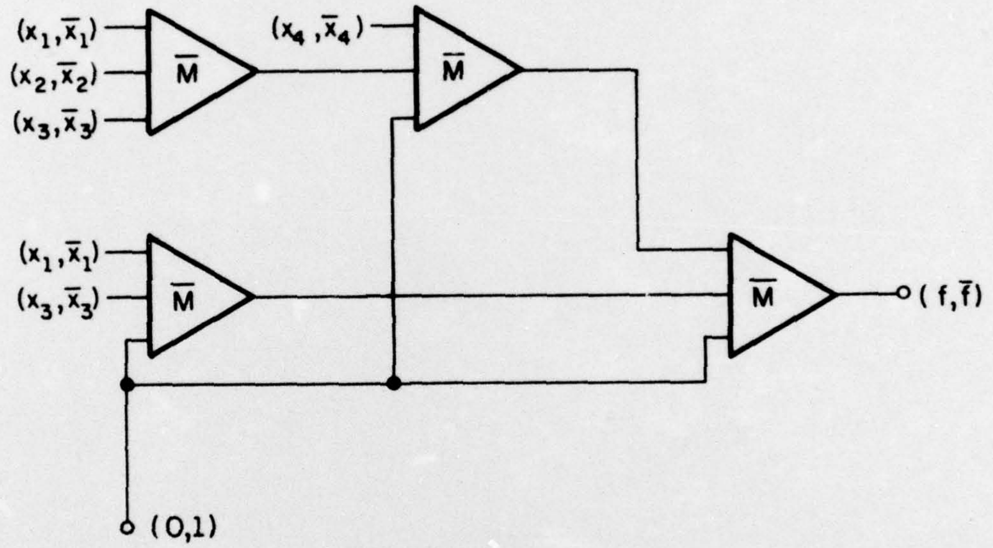
Utilize the $f^{(1)}$ function where possible as a subfunction in f.

$$f = \overline{\overline{x_4 f^{(1)}(x_1,x_2,x_3)}\ \overline{x_1x_3}}$$

The resulting minority logic alternating network is shown in Figure 7.5.

A similar problem exists for the majority logic alternating design procedure discussed in the previous section. No effort is made to utilize the M-module as a basic gate and minimality of the alternating design obtained can therefore not be guaranteed even if the conventional AND/OR/NOT design is optimal.

A minority logic alternating design procedure can also be defined utilizing alternating NOR's. This procedure is the dual of the design



FP-5079

Figure 7.5. Minority logic alternating network.

procedure defined for minority logic alternating design using alternating NAND. As such it begins with a conventional NOR decomposition of the function to be realized. Each NOR is then replaced by an alternating NOR, so the (1,0) clock is required and primary network inputs become primary alternating inputs synchronized with this clock. Comments on optimality apply here as in the case of the alternating NAND design discussed earlier.

7.5. Self-Checking Characteristics of Compositions of Alternating Logic Primitives

The methods which have been described yield alternating circuits composed of alternating logic primitives representing any arbitrary Boolean function. These circuits, however, may not be self-checking. Consider an arbitrary alternating network N constructed as a composition of alternating logic primitives with n inputs $(x_1,\overline{x}_1),(x_2,\overline{x}_2),\ldots,(x_n,\overline{x}_n)$ and m outputs $(f_1,\overline{f}_1),(f_2,\overline{f}_2),\ldots,(f_m,\overline{f}_m)$. During fault-free operation inputs alternate and, as a result, the output of any logic primitive having only primary network inputs as inputs must alternate. But then, the output of any logic primitive having as inputs, primary network inputs or outputs from primitives having as input only primary network inputs, must alternate. Continuing, the output of every alternating logic primitive in a composition of alternating logic primitives must alternate if alternating inputs are applied. Faults, however, may occur in the network. Such faults are assumed to occur as stuck-at faults on input and output lines of alternating logic primitives. More formally, the fault set $\mathfrak{F}$ is assumed to be the set of all single stuck-at-0 and stuck-at-1 faults on any alternating logic primitive input or output line. Assuming that a fault from the set $\mathfrak{F}$ is present in the network N, the network is self-checking for that fault if no erroneous alternating output appears on f_i, $1 \leq i \leq m$, unaccompanied by a nonalternating output on some f_j, $1 \leq j \leq m$, $i \neq j$.

Theorem 7.5: Any network N consisting of a composition of alternating logic primitives is self-checking for all single stuck-at-0 and stuck-at-1 faults on any alternating logic primitive input or output line.

Proof: Assume a single fault f_a^d is present on an arbitrary logic primitive input or output line, a. Since the network N is a composition of alternating logic primitives, in fault-free operation either $(d,\overline{d})$ or $(\overline{d},d)$ is applied on a. If $(d,\overline{d})$ is applied then the fault is sensitized only during the application of $\overline{X} = \overline{x}_1\overline{x}_2\ldots\overline{x}_n$ to N. So, on any output f_i, $1 \leq i \leq m$, either (f_i,f_i) is produced if $\overline{X}$ also sensitizes a path from a to output f_i, or $(f_i,\overline{f}_i)$, the normal output, is produced if $\overline{X}$ does not sensitize a path from a to the output f_i. If $(\overline{d},d)$ is applied then the fault is sensitized only during the application of $X = x_1x_2\ldots x_n$ to N. As a result, on any output f_i, $1 \leq i \leq m$ only $(f_i,\overline{f}_i)$ or $(\overline{f}_i,\overline{f}_i)$ is produced. In no case can an erroneous alternating output be produced and N is therefore self-checking for all single faults from the fault set $\mathfrak{F}$. Q.E.D.

The significance of Theorem 7.5 lies in its generality. According to the theorem any alternating network resulting from the minority and majority logic alternating design procedures must be self-checking, as the procedures yield alternating networks which are structured as compositions of alternating logic primitives. In fact, these networks are self-checking irregardless of the network structure utilized in the conventional designs which initiate the design procedures. In Example 7.2 the conventional NAND network contained reconvergent fanout with unequal inversion parity on output paths originating on the fanout point. And, while the alternating network resulting from the application of the design procedure retains the same structure, Theorem 7.5 says that it must be self-checking for all single faults.

8. CONCLUSION

8.1. Summary

The application of alternating logic in the design of combinational and sequential circuits with the capability to detect all single stuck-at faults on all network lines without passing erroneous data to the user has been studied. The fault detecting capability was obtained by utilizing a redundancy in time instead of the conventional space redundancy and is based on the successive execution of a required function and its dual. In combinational networks the method involves the utilization of a self-dual function to represent the required function and the realization of the self-dual function in a network with structural properties which are sufficient to guarantee the detection of all single faults. The internal fanout-free network, the essentially inverter-free AND/OR/NOT network, and the inverter-free network were defined and shown to be network structures having these properties. Further, necessary and sufficient conditions that any general alternating combinational *structure be self-checking* were derived. That a network satisfies these conditions was shown to be readily checked utilizing computer simulation. Multi-output combinational structures were also discussed and an important multi-output network, the full adder, was studied in detail. With respect to this network, the advantages of sharing logic in multi-output alternating networks were illustrated.

A particular class of combinational alternating networks, namely NAND/NOR alternating networks were examined in more detail. An algebra for manipulating the functional description for the alternating NAND/NOR network for the purpose of obtaining a more economical realization was developed. The algebra consisted of a set of transformations, which when applied to a functional description of a self-checking alternating NAND/NOR network, could only result in an alternate, but hopefully more economical, functional description while retaining the self-checking structural properties. The method was applied to the full-adder, beginning with its standard NAND two-level functional description and generating its optimal NAND multilevel realization.

The cost of implementing the dualized function as a self-checking alternating network as compared with the cost of implementing the original function in a conventional design was also studied. The study was restricted to NOR networks and the cost of dualizing all three-variable functions was studied. Only NOR networks and three-variable functions were considered because of availability of data. The results showed that on the average a cost increase of 85% resulted. This, however, did not include the cost of a checker for monitoring output lines.

The application of alternating logic design to synchronous sequential machines was also studied, and a method for realizing any synchronous sequential machine with feedback and output circuitry designed using combinational alternating logic techniques was presented. The resulting realization was shown to be capable of detecting all single faults in the feedback and output combinational circuitry and all single input and output faults associated with the memory elements. While the method did not place any restraints on the state assignment of the secondary variable, the number of memory elements required was doubled. However, the extra set of memory elements added also served as the memory elements required by the alternating checker and therefore reduced cost increase there.

Since some means must be provided to monitor network output lines (in the case of synchronous machines, network output lines and feedback lines) so as to provide an indication of the occurrence of nonalternating outputs on these lines, a network performing this task, the alternating checker, was defined. One such checker, utilized a memory element for each monitored line and performed the self-checking function in the general case while being self-checking itself. A second checker utilized only exclusive-OR gates in its construction and performed the self-checking function on networks possessing the property that nonalternating outputs occur on only a single line at a time. This checker was also shown to be self-checking.

An alternate approach to designing self-checking alternating networks was also studied. Instead of utilizing conventional logic primitives (AND/OR/NOT, NAND, NOR, etc.), self-dual logic primitives were considered since composing these primitives to form a network naturally yields an alternating network. Conditions that a set of these primitives be complete, that is, that all Boolean functions be represented as a composition of these primitives, were defined. Further, it was shown that any network, regardless of its structure, is self-checking for all single stuck-at faults on alternating primitive input and output lines. The alternating primitive set, {M,NOT}, where M is a three-input majority gate, was studied in detail as it allowed any conventional AND/OR/NOT network design to be converted into a self-checking network of M gates and NOT gates with particular ease.

With respect to the area of applicability of alternating logic, some remarks should be made. First, since alternating systems utilize a redundancy in time in order to achieve fault detection capabilities, an alternating system approach to fault detection probably should not be used where time is a constraining factor. However, in clocked systems where logic speed is more than double the speed required by the application, alternating logic design can be an effective technique for producing combinational and sequential circuits which are self-checking for all single faults. Utilization of redundancy in time, however, also leads to an advantage in that dynamic checking of circuit operation is achieved without an increase in the number of input and output lines associated with the network itself. Thus, in MSI and LSI applications where pin count is a major factor, an alternating logic design could be a feasible approach in achieving a measure of fault detecting capability.

8.2. Suggestions for Further Research

The capability of alternating systems to detect all single faults of the stuck-at type while not passing erroneous data to the user has been studied. Specific alternating systems, however, may have some multiple fault detection capabilities. These multiple fault detection capabilities, if any, need to be researched and defined.

In order to gain a measure of multiple fault detection capability, it may however be necessary to utilize a more complex coding scheme than the basic alternating scheme studied here. The systems studied here in essence employ a two-rail encoding in time rather than in space and hence are referred to as alternating systems. The encoding used, however, may be generalized to more complex time encodings. One result of this could be a reduction in the ratio of the number of check bits to information bits, which for alternating systems is 1/1. Such a reduction would, of course, alleviate at least to some extent some of the speed restrictions that limited the applicability of alternating systems. A second result could be the multiple fault detection capability mentioned earlier. Timing considerations could, however, be prohibitive.

The application of alternating logic to synchronous sequential machines has been studied. It may also be possible to apply alternating logic to some forms of asynchronous machines, particularly those machines which employ a "handshaking" scheme. Research in this area might also prove fruitful.

REFERENCES

1. Anderson, D. A., "Design of Self-Checking Digital Networks Using Coding Techniques," Coordinated Science Laboratory Report R-527, University of Illinois, September 1971.

2. Anderson, D. A. and Metze, G., "Design of Totally Self-Checking Check Circuits for m-out-of-n Codes," IEEE Trans. on Computers, Vol. C-22, pp. 263-269, March 1973.

3. Avizienis, A., Gilley, G. C., Mathur, F. P., Rennels, D. A., Rohr, J. A., and Rubin, D. K., "The STAR (Self-Testing-and-Repairing) Computer: An Investigation of the Theory and Practice of Fault Tolerant Computer Design," Digest 1971 International Symp. on Fault-Tolerant Computing, Pasadena, pp. 92-96, March 1971.

4. Bark, A. and Kinne, C. B., "The Application of Pulse Position Modulation to Digital Computers," Proc. National Electronics Conf., pp. 656-664, September 1953.

5. Berger, J. M., "A Note on Error Detection Codes for Asymmetric Channels," Information and Control, Vol. 4, pp. 68-73, March 1961.

6. Beuscher, H. J., Fessler, G. E., Huffman, D.W., Kennedy, P. J., and Nussbaum, E., "No. 2 ESS Administration and Maintenance Plan," Bell System Tech. Journal, Vol. 48, pp. 2765-2815, October 1969.

7. Carter, W. C. and Bouricius, W. G., "A Survey of Fault-Tolerant Architecture and Its Evaluation," Computer, Vol. 4, pp. 9-16, January/February 1971.

8. Carter, W. C., Bouricius, W. G., Jessep, D. C., Roth, J. P., Schneider, P. R., and Wadia, A. B., "A Theory of Design of Fault-Tolerant Computers Using Standby Sparing," Digest 1971 International Symp. on Fault-Tolerant Computing, Pasadena, pp. 83-86, March 1971.

9. Carter, W. C. and Schneider, P. R., "Design of Dynamically Checked Computers," IFIP 68, Vol. 2, Edinburg, Scotland, pp. 878-883, August 1968.

10. Carter, W. C., Jessep, D. C., and Wadia, A. B., "Error-Free Decoding for Failure-Tolerant Memories," Proc. 1970 IEEE International Computer Group Conf., Washington, D. C., pp. 229-239, June 1970.

11. Chang, H. Y., Manning, E., and Metze, G., Fault Diagnosis of Digital Systems, Wiley-Interscience, New York, 1970.

12. Davidson, E. S., "An Algorithm for NAND Decomposition of Combinational Systems," Coordinated Science Laboratory Report R-382, University of Illinois, May 1968.

13. Diaz, M., Geffroy, J. C., and Courvoisier, M., "On-Set Realization of Fail-Safe Sequential Machines," IEEE Trans. on Computers, Vol. C-23, pp. 133-138, February 1974.

14. Diaz, M., "Design of Totally Self-Checking and Fail-Safe Sequential Machines," Fourth Annual International Symp. on Fault-Tolerant Computing, pp. 3-19 to 3-24, June 1974.

15. Downing, R. W., Nowick, J. S., and Tuomenoksa, L. S., "No. 1 ESS Maintenance Plan," Bell System Tech. Journal, Vol. 43, pp. 1961-2019, September 1964.

16. Friedman, A. D. and Menon, P. R., Fault Detection in Digital Circuits, Prentice-Hall, 1971.

17. Hayes, J. P., "A Study of Digital Network Structure and its Relation to Fault Diagnosis," Coordinated Science Laboratory Report R-467, University of Illinois, May 1970.

18. Hellerman, L., "A Catalog of Three-Variable OR-Invert and AND-Invert Logical Circuits," IEEE Trans. on Electronic Computers, Vol. EC-12, pp. 198-223, June 1963.

19. Hohn, F. E., "Algebraic NAND-NOR Logic Design, an Exposition," Coordinated Science Laboratory Report T-20, University of Illinois, August 1975.

20. Ibuki, K., Naemura, K., and Nozaki, A., "General Theory of Complete Sets of Logical Functions," Electronics and Communications in Japan (IEEE translation), Vol. 46, No. 7, pp. 55-65, July 1963.

21. Kohavi, Z., Switching and Finite Automata Theory, McGraw-Hill, New York, 1970.

22. Liu, T., Hohulin, K. R., Shiau, L., and Muroga, S., "Optimal One-Bit Full Adders with Different Types of Gates," IEEE Trans. on Computers, Vol. C-23, No. 1, January 1974.

23. Mukai, Y. and Tohma, Y., "A Method for the Realization of Fail-Safe Asynchronous Sequential Circuits," IEEE Trans. on Computers, Vol. C-23, pp. 736-739, July 1974.

24. Mukhopadhyay, A., "Complete Sets of Logic Primitives," Recent Developments in Switching Theory, Academic Press, New York, pp. 1-26, 1971.

25. Nakagawa, T. and Lai, H., "Reference Manual of Fortran Program Illod-(NOR-B) for Optimal NOR Networks," Digital Computer Laboratory Report R-71, University of Illinois, December 1971.

26. Ozguner, F., "Design of Totally Self-Checking Asynchronous Sequential Machines," Coordinated Science Laboratory Report R-679, University of Illinois, May 1975.

27. Patterson, W. W., "On the Design of Diagnosable Asynchronous Sequential Machines," Coordinated Science Laboratory Report R-525, University of Illinois, September 1971.

28. Reddy, S. M., "A Note on Self-Checking Checkers," <u>IEEE Trans. on Computers</u>, Vol. C-23, pp. 1100-1102, October 1974.

29. Reddy, S. M. and Ashjaee, M. J., "Totally Self-Checking Checkers for a Class of Separable Codes," <u>Twelfth Annual Allerton Conf. on Circuit and System Theory</u>, pp. 238-240, October 1974.

30. Reddy, S. M. and Wilson, J. R., "Easily Testable Cellular Realizations for the (Exactly P)-out-of-n and (P or More)-out-of-n Logic Functions," <u>IEEE Trans. on Computers</u>, Vol. C-23, pp. 98-100, January 1974.

31. Sawin, D. H., "Fail-Safe Synchronous Sequential Machines Using Modified On-Set Realizations," <u>Fourth Annual International Symp. on Fault-Tolerant Computing</u>, pp. 3-7 to 3-12, June 1974.

32. Sellers, F. F., Hsiao, M. Y., and Bearnson, L. W., <u>Error Detecting Logic for Digital Computers</u>, McGraw-Hill, New York, 1968.

33. Smith, J. E. and Metze, G., "General Design Rules for the Construction of m-out-of-n Totally Self-Checking Checkers," Coordinated Science Laboratory Report R-693, University of Illinois, October 1975.

34. Tohma, Y., Ohyama, Y., and Royza, S., "Realization of Fail-Safe Sequential Machines by Using a k-out-of-n Code," <u>IEEE Trans. on Computers</u>, Vol. C-20, pp. 1270-1275, November 1971.

35. Wang, L. L. and Chuang, H. Y. H., "On the Improvement of Fail-Safe Synchronous Machine Design Using On-Set Realization," <u>Digest 1975 International Symp. on Fault-Tolerant Computing</u>, June 1975.

36. Yamamoto, H., Watanabe, T., and Urano, Y., "Alternating Logic and its Application to Fault Detection," <u>Proc. 1970 IEEE International Computer Group Conf.</u>, Washington, D. C., pp. 220-228, June 1970.

VITA

Dennis Andrew Reynolds was born in El Dorado, Arkansas on November 9, 1947. He received the BSEE degree from the University of Arkansas, Fayetteville, in 1969 and the MSEE degree from the University of Illinois, Urbana-Champaign, on Sandia Laboratories One Year on Campus (OYOC) Program in 1970. Currently, he is in the Ph.D. program of the Electrical Engineering Department of the University of Illinois under Sandia Laboratories Doctoral Study Program (DSP). Since 1969 he has been a member of the Technical Staff at Sandia Laboratories, Albuquerque, New Mexico, where he has been concerned with high speed A/D conversion and computer controlled systems. Mr. Reynolds is a member of Tau Beta Pi and Eta Kappa Nu.